DOCUMENT RESUME

ED 255 207 IR 011 579

TITLE	Computer Science 205. Interim Guide, 1983.
INSTITUTION	Manitoba Dept. of Education, Winnipeg.
PUB DATE	83
NOTE	274p.; For related documents, see IR 011 580-582.
PUB TYPE	Guides - Classroom Use - Guides (For Teachers) (052)
EDRS PRICE	MF01/PC11 Plus Postage.
DESCRIPTORS	Computer Assisted Instruction; Computer Literacy; *Computers; *Computer Science Education; Course Objectives; Curriculum Guides; Foreign Countries; Learning Activities; *Problem Solving; *Programing; *Programing Languages; Secondary Education
IDENTIFIERS	*Manitoba

ABSTRACT

This guide to a 4-unit, required high school computer science course emphasizes problem solving and computer programming and is designed for use with a variety of hardware configurations and programming languages. An overview covers the program rationale, goals and objectives, program design and description, program implementation, time allotment, language requirement, format and sequencing, hardware, software, textbooks, and student evaluation. Objectives and suggested learning activities are listed for each unit. The first unit introduces the components of computer systems and their interrelationships, hardware and software development, and the use of stored programs. Unit 2 stresses the development of general problem solving techniques using algorithms expressed in flowcharts or psuedo-code. Unit 3 covers programming in psuedo-machine language, psuedo-assembly language, and high-level language. The final unit on programming techniques explains debugging and characteristics of a good program. Appendices contain information on spectre opcodes, structured programming, conditional statements, and suggestions for assignments. (LMM)

****

* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *

****

IR011579

ED255207

COMPUTER SCIENCE 205

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION
CENTER (ERIC)

☒ This document has been reproduced as received from the person or organization originating it.

Minor changes have been made to improve reproduction quality.

- Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

Interim Guide

1983

PROVINCE OF MANITOBA

DEPARTMENT OF EDUCATION

THIS GUIDE IS SUBJECT TO REVISION ON THE BASIS OF INFORMATION RECEIVED FROM SCHOOLS. IT IS EXPECTED THAT MODIFICATIONS WILL NOT BE IN THE DESIGN OF THE PROGRAM. RATHER, CHANGES MAY BE MADE IN AREAS CONCERNING STRATEGIES, RESOURCE MATERIALS, AND IMPLEMENTATION.

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
Keith A. Cosens

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

ISBN 0-7711-0144-9

ACKNOWLEDGEMENTS

Sincere appreciation is extended to the Computer Science 205/305 Committee, which developed the program and prepared the curriculum guidelines for Computer Science 205.

Members of the Committee:

Tom Blair
Joseph Wolinsky Collegiate
Private and Parochial School Division No. 99

Allison Challes
St. Mary's Academy
Private and Parochial School Division No. 99

Bob Finlayson
Neepawa Area Collegiate
Beautiful Plains School Division No. 31

Bob Fraser
Crocus Plains Regional Secondary School
Brandon School Division No. 40

George McMaster
Computer Science Department
Brandon University

Margaret McRitchie (until October, 1980)
Kildonan East Regional Secondary School
River East School Division No. 9

Jim Pascoe (until November, 1981)
Sisler High School
Winnipeg School Division No. 1

Rob Rummery
St. James Collegiate
St. James-Assiniboia School Division No. 2

Ken Schmidt
Crocus Plains Regional Secondary School
Brandon School Division No. 40

Frank Wilson (until March, 1982)
Vincent Massey High School
Fort Garry School Division No. 5

Consultants to the Committee:

Duncan McCaig
and
Wayne Watt
Curriculum Development and Implementation Branch
Department of Education

Appreciation is also extended to the many computer science teachers who provided input to the committee and in particular to the following teachers who participated in piloting the guide.

Chris Boumford
Shaftesbury High School
Assiniboine South School Division No. 3

John Fast
River East Collegiate
River East School Division No. 9

John Fotheringhame
Neelin High School
Brandon School Division No. 40

Lawrie Goalen
Edward Schreyer School
Agassiz School Division No. 13

Randy Greenlay
Lord Selkirk Regional Secondary School
Lord Selkirk School Division No. 11

Bill Korytowski
Sisler High School
Winnipeg School Division No. 1

Dave Owczar
Glenboro Collegiate
Tiger Hills School Division No. 29

Kevin Rowan
Fort Richmond Collegiate
Fort Garry School Division No. 5

Wayne Stowman
Margaret Barbour Collegiate
Kelsey School Division No. 45

Harry Yoshida
Shaftesbury High School
Assiniboine South School Division No. 3

7

8

CONTENTS

PROGRAM OVERVIEW

Rationale 3
Goals and Objectives 3
Program Design and Description 6
Program Implementation 6
Time Allotment 7
Language Requirement 7
Format and Sequencing 7
Hardware 8
Software 8
Textbooks 8
Student Evaluation 8

COURSE OUTLINE

UNIT I - INTRODUCTION (10%)

A. Components of the Computer 13
B. Hardware and Software 22
C. Stored Programs 25

UNIT II - PROBLEM SOLVING (20%) 26

UNIT III - PROGRAMMING (60%)

A. Pseudo-Machine Language 52
B. Pseudo-Assembly Language 66
C. High-Level Language 75

UNIT IV - TECHNIQUES (10%)

A. Debugging 114
B. Characteristics of a Good Program 119

APPENDICES

A. Spectre Opcodes 125
B. Structured Programming 127
C. Conditional Statements 134
D. Suggestions for Tests or for Assignments . . . 137
E. Suggestions for Use of Program Comments . . . 139

program overview

PROGRAM OVERVIEW

RATIONALE

Over the past decade major societal and technological advances have considerably altered the requirements for computer education in Manitoba high schools. As a result of those advances, and of recent hardware and language developments, a complete redesign of Computer Science 205/305 has proven necessary. The new outline allows for the use of micro systems, mini systems and mainframe systems including the Manitoba High School Network as the hardware support for course implementation. In addition, the new outline allows choice of high-level languages to permit flexibility in system selection.

It is recommended that at least two high-level languages, selected from among BASIC, Pascal, FORTRAN, and COBOL, be taught in Computer Science 205/305. To aid in the understanding of how the computer operates on high-level language programs, pseudo-machine language and pseudo-assembly language have been introduced in Computer Science 205.

Because of the increase in the number of programming personnel and rapid changes in the computer field, it is necessary to standardize programming techniques. The trend in recent years has been toward structured programming. A major thrust of this new outline is therefore to emphasize and to relate structured programming techniques. For additional information refer to Appendix B, "Structured Programming" on page 127.

In the last few years there has been a significant divergence in the approach and content of high school computer science. The Committee intends the new program to allow for individual hardware-related differences and at the same time encourages standardization of course content across the province.

GOALS AND OBJECTIVES

Goals

Computer Science 205 has been designed so as to support the achievement of students in:

1. recognizing the basic features and functions of different computer configurations;
2. developing problem-solving techniques using algorithms, which are expressed as flowcharts or pseudo-code;
3. making appropriate use of computers as tools for solving problems;
4. utilizing effective structured programming techniques in problem-solving;
5. understanding the operation of a computer as it utilizes pseudo-machine and pseudo-assembly language programs;
6. understanding the interrelation of pseudo-machine, pseudo-assembly and high-level language;
7. preparing and executing structured programs written in a high-level language;
8. developing facility in the detection and correction of programming errors; and
9. employing accepted programming techniques.

Objectives

Unit 1 - Introduction (10%)

A. The Components of the Computer

The student should be able to:

1. identify the five main components of a computer and the interrelationship of each component;
2. identify the devices used with each component;
3. identify factors affecting the selection of input and output devices;
4. identify factors affecting the selection of storage devices.

B. Hardware and Software

The student should be able to:

1. distinguish/define hardware, software and firmware;
2. discuss various computer systems according to cost, memory and speed.

C. Stored Programs

The student should be able to execute a "canned" program given appropriate documentation.

Unit II - Problem Solving (20%)

The student should be able to:

1. discuss various methods used in solving problems;
2. identify the logical series of steps involved in solving a problem;
3. define the term "algorithm";
4. define the terms "flowchart" and "pseudo-code";
5. correctly identify and understand the meaning of the flowchart symbols;
6. recognize three types of logical steps or constructs:
   (a) linear (sequential),
   (b) branching (IF. . . THEN . . . ELSE . . .),
   (c) looping;
7. graphically represent the solution to simple non-numeric and numeric problems using:
   (a) flowchart,
   (b) pseudo-code;
8. test his/her algorithm.

Unit III - Programming (60%)

A. Pseudo-Machine Language - 5%

The student should be able to:

1. describe the characteristics of the five basic parts of a computer as they relate to a pseudo-machine language;
2. identify the following pseudo-machine language statements:
   (a) load statements,
   (b) control statements;
3. develop programs using the load and control statements;
4. run a pseudo-machine language program on a computer;
5. identify and describe the two phases involved in processing a program, including:
   (a) load,
   (b) run;
6. interpret error diagnostics involving:
   (a) load time,
   (b) run time;
7. write simple programs using input statements and run-time data;
8. write programs using the following constructs:
   (a) sequence,
   (b) looping,

(c) branching.

B. Pseudo-Assembly Language - 5%

The student should be able to:

1. identify and use the necessary control commands for a pseudo-assembly language program;
2. run a pseudo-assembly language program on a computer;
3. translate a pseudo-assembly language to pseudo-machine language;
4. identify and use literals at load time;
5. describe the operation of the assembler (i.e., load time has two passes).

C. High-Level Language - 50%

The student should be able to:

1. draw analogies between computer programs and everyday tasks;
2. identify and use the fundamental statements for:
   (a) documentation,
   (b) assignment,
   (c) output,
   (d) input,
   (e) termination;
3. describe and show the application of the conditional statement;
4. identify and use various types of looping techniques;
5. identify and use formatted input and output;
6. identify and use the CASE statement;
7. identify the need for simple data structures and develop and use simple data structures involving:
   (a) an appreciation of the trend in computer programming, from an emphasis on numerical computation to an emphasis on the storage and retrieval of data,
   (b) consideration of the problem of what data best represents the information that is to be processed and then how to represent that data on the computer,
   (c) recognition that there are many ways of representing data, even simple non-structured data such as one student mark,
   (d) choosing a data structure dependent on the operations that are to be performed on that data, making the problem easier to solve,
   (e) description of how an array is stored internally, and how an individual element is accessed,
   (f) recognition of how a one-dimensional array makes the solution to some problems possible and makes other problems easier to solve,
   (g) appreciation that a particular data structure permits certain operations to be carried out efficiently,
   (h) recognition of the need for a two-dimensional array, and
   (i) description of how a two-dimensional array is stored internally;
8. create, manipulate and retrieve files.

Unit IV - Techniques - 10%

A. Debugging

The student should be able to:

1. detect and correct programming errors involving syntax;
2. detect and correct programming errors in logic.

B. Characteristics of a Good Program

The student should be able to:

1. identify good programming techniques;
2. use good programming techniques.

PROGRAM DESIGN AND DESCRIPTION

This course has been developed with major emphasis on problem solving and computer programming. The intent is for the student to use the computer as a means of acquiring problem-solving skills. Techniques for problem solving learned in this way will also be applicable in disciplines other than computing. Throughout the instruction and practice, good programming style must be emphasized.

Computer components and terms are defined, and the limitations and advantages of different types of computers are presented. The Computer Science 205 guide has been designed for use with a variety of hardware configurations and has been written with BASIC, Pascal, and FORTRAN languages in mind. Students do not require prior knowledge or experience with computers before beginning this course.

Computer Science 205 is comprised of four major units emphasizing programming and problem-solving techniques. The units are as follows:

Unit I - Introduction (10%)

This unit deals with the components of computer systems and their interrelationships, hardware and software development, and the use of stored programs. In schools offering Computer Awareness 105 this unit may be largely review.

Unit II - Problem Solving (20%)

The development of general problem-solving techniques using algorithms expressed in flowcharts or pseudo-code is stressed. Since all problems can be solved using the three constructs of sequencing, branching and looping, these are dealt with extensively in a structured format.

Unit III - Programming (60%)

The overall purpose of the course is to learn to program the computer in a high-level language and to apply good problem-solving techniques in the development of these programs. To impart an understanding of how the computer carries out high-level language statements, it is recommended that 10% of the course time be spent on pseudo-machine and pseudo-assembly language programming (SPECTRE and SPECTRE MAP), and that 50% be spent on high-level language programming.

Unit IV - Programming Techniques (10%)

Since creating computer programs without errors is a rarity, it is essential that students learn to locate errors, correct them, and rigorously test their program(s). Also, since programs in "real life" situations are usually modified after they are initially introduced, it is necessary to develop good programming techniques. Structured programming techniques will make it easier to see the logic used in a program, and also will allow modifications to be made with relative ease.

PROGRAM IMPLEMENTATION

This Computer Science 205 curriculum has been designed to be hardware and software independent. Two major differences between the old and new curriculum are:

1. the inclusion of a pseudo-machine and pseudo-asembly language, and
2. emphasis on structured programming constructs and techniques.

TIME ALLOTMENT

As a one-credit course, Computer Science 205 requires 110-120 hours of instruction time. All schools/school divisions are expected to ensure that sufficient time is allocated to provide for the achievement of the objectives outlined. This time should be broken down as follows:

Unit I - Introduction - 10%
Unit II - Problem Solving - 20%
Unit III - Programming
    (a) Pseudo-Machine Language - 5%
    (b) Pseudo-Assembly Language - 5%
    (c) High-Level Language - 50%
Unit IV - Techniques - 10%

LANGUAGE REQUIREMENT

Schools teaching Computer Science 205 may use any high-level language such as BASIC, FORTRAN, or Pascal. However, it is important to note that if schools wish to teach Computer Science 305, a second language other than the one chosen in 205 must be used. The Committee suggests that BASIC or FORTRAN be taught in 205 and that Pascal or COBOL be taught in 305.

FORMAT AND SEQUENCING

Although the course outline consists of four units, it is not intended that these units be taught either as separate units or in the order presented. Teachers are expected to select an appropriate order of content presentation based on the needs of their students. For the benefit of teachers wishing to integrate units or parts of units, the following chart indicates related pages from this guide for each of the structured topics.

	LINEAR	BRANCHING IF...THEN ...ELSE	LOOPING	CASE
PROBLEM SOLVING	31	31-32	33-34	35
PSEUDO-CODE FLOWCHARTING	36-37	38-39	40-44	45-46
PROGRAMMING	62	64-65 80-84	62, 85-91	93-98

Many possibilities exist for order of presentation. Three such possibilities are:

1. (a) Introduction - 10%
   (b) Pseudo-Machine-Assembler - 10%
   (c) High-Level Language - 80%
      (i) Linear Sequence
         (a) Problem Solving

        - (b) Flowcharting - Pseudo-Code
        - (c) Programming Problems
        - (d) Debugging
    - (ii) Branching (IF...THEN...ELSE)
        - (a) Problem Solving
        - (b) Flowcharting - Pseudo-Code
        - (c) Programming Problems
        - (d) Debugging
    - (iii) Looping (WHILE DO, REPEAT UNTIL)
        - (a) Problem Solving
        - (b) Flowcharting - Pseudo-Code
        - (c) Programming Problems
        - (d) Debugging
    - (iv) Data Structures

2. (a) Introduction - 10%
    - (b) Problem Solving - 20%
    - (c) Pseudo-Machine and Pseudo-Assembly Languages - 10%
    - (d) High-Level Language and Techniques - 60% (basically follows order of outline)

3. (a) Introduction - 10%
    - (b) Problem Solving - 20%
    - (c) High-Level Language and Techniques - 50%
    - (d) Pseudo-Machine and Pseudo-Assembly Languages - 10%
    - (e) Data Structures - 10%

HARDWARE

The Computer Science 205 curriculum has been designed to be hardware independent and the objectives may be achieved using micros, mini and/or network systems. A major consideration in determining hardware requirements is the scheduling of a minimum of 40 hours per student of machine-contact time (this contact time does not necessarily have to be during regular class time). A good ratio to fulfill the "hands-on" requirement for this course would be one interactive terminal for every two students in a particular class.

SOFTWARE

Although the Computer Science 205 guide has been designed for network, mini and micro equipment, software support is available for only network, Apple II, PET, and TRS-80 users. Schools requiring a diskette copy of SPECTRE should forward a blank diskette indicating type of microcomputer to Computer Services, Department of Education.

TEXTBOOKS

Because schools are using different languages and different equipment, and because textbooks in the computer field quickly become outdated, a single student textbook has not been selected for Computer Science 205.

Books which could be used as classroom texts will, however, be identified in the "Recommended Supplementary Materials" section of the Manitoba Text Book Bureau Catalogue.

STUDENT EVALUATION

Prior to student evalution, it is important that the teacher specify to students the course content, the methods of evaluation to be used, and the approaches students should take in fulfilling the course requirements.

In selecting methods of evaluation, teachers should keep in mind the fact that Computer Science 205 is basically a problem-solving and programming course. Specific factors relevant to evaluation include:

1. the quantity of hardware available. Take this into consideration when assigning numbers of questions to be run on a computer;
2. the fact that students can easily copy another student's program and, therefore, more emphasis should be put on tests rather than assigned programs.

In addition to giving the usual assignment of constructing a program and running it, the teacher should consider:

1. having the student read a program, describe what it does, document it, run it and generate the output given specific input;
2. modifying a given program in order for the program to do something different;
3. handing out a program containing errors and having students correct them.

Actual evalution could consist of:

1. Program assignments

   Consider weighting questions with most of the marks allocated to the first few questions so that if questions are not completed in the time-frame allowed, students can still pass.

2. Program tests

   (a) constructing a program for a particular problem
   (b) correcting programs with errors
   (c) documenting a given set of code

3. Definitions

4. True/false and multiple-choice tests.

See Appendix D for further suggestions and examples concerning tests and assignments.

course outline

COMPUTER SCIENCE 205
COURSE OUTLINE

UNIT I. INTRODUCTION

A. The Components of the Computer

OBJECTIVES	NOTES AND EXPLANATION
The student should be able to: 1. identify the main components of a computer and the interrelationship of each component;	Computer components to be considered are: 1. Input Unit - piece of equipment through which data and instructions are entered into the computer Types a) batch (where several jobs can be submitted to the computer at the same time) Example: card reader b) interactive (where the programmer uses a console to communicate with the computer on a one-to-one basis) Examples: CRT - cathode ray tube - also known as video display terminal (VDT); teletype terminal; microcomputers c) specialized (less human involvement required in data entry) Examples: (i) light sensitive devices which pick up codes such as the universal product code in grocery stores or the postal code on mail

OBJECTIVES | NOTES AND EXPLANATION

(ii) touch sensitive - touch sensitive screens; cash registers
(iii) voice sensitive - security systems, speech recognition systems
(iv) magnetic sensitive - cheques, credit cards

2. Output Unit

- piece of equipment through which the processed information passes to the user

Examples:
- printers (character, line, page). Printing mechanisms could include chain, wheel, bar, drum, matrix, laser, or ink jet.
- T. V. screens
- tape drives
- disk drives
- cash registers and tapes at computerized checkouts
- voice - audio response units (e.g., SPEAK 'N SPELL)
- graph plotters
- card punches
- paper tape punches

3. Central Processing Unit (CPU)

- usually consists of a Control Unit, a Memory Unit, and an Arithmetic-Logic Unit

a) Control Unit

- interprets instructions and uses a timing device to control the operation and interaction of all the other parts of the computer

OBJECTIVES

NOTES AND EXPLANATION

b) Memory Unit - (primary storage)

- stores all data and instructions in a binary code consisting of a combination of zeros and ones

c) Arithmetic-Logic Unit

- performs mathematical operations and makes decisions. The computer performs its decision-making by comparing two numbers to see whether one is less than, equal to, or greater than the other.

Many computers have other devices, attached to their C.P.U.'s, which are capable of storing information. This is often necessary since the C.P.U. memory might be too small or because the information might have to be retained over a period of time. Such storage is divided into two categories.

(i) Auxiliary Storage (Secondary Storage)

- connected to the C.P.U.

Examples: tape drive units (9-track, 7-track, cassettes); disk drive units (hard, floppy)

(ii) External Storage

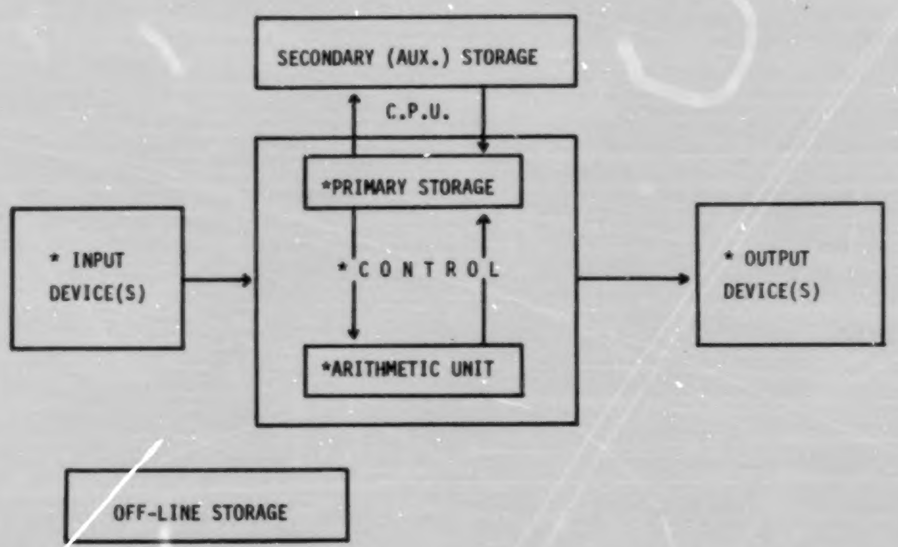
- separate and not connected with the C.P.U.

Examples: libraries where disks or tapes, which have been removed from the devices listed above, are stored

Points to Emphasize

- All computers regardless of size have the five components (input, output, storage, control and arithmetic-logic).

OBJECTIVES	NOTES AND EXPLANATION
	. The five components may be in one physical unit or in several separate units . The relationship of the components can be seen in the following diagrams:


SECONDARY (AUX.) STORAGE
C.P.U.
*PRIMARY STORAGE
* INPUT DEVICE(S)
*CONTROL
* OUTPUT DEVICE(S)
*ARITHMETIC UNIT
OFF-LINE STORAGE

NOTE: Storage has been shown as primary, secondary, and off-line.

* denotes that this component is one of the five main components of a computer

Suggestions for Presentation

1. Compare computer opertion to a person performing arithmetic.

    a) Before the calculation can be done one must recall what operation is being attempted (i.e., instruction).
    b) The numbers (data) being used are written down (i.e., data is entered into storage).
    c) These numbers are then processed according to the rules of arithmetic (i.e., the arithmetic-logic unit processes the data).
    d) The resulting answer is then written (i.e., the answer is transferred to output).

2. Have the student consider the following typical classroom problem.

    The teacher asks a student to multiply two numbers together and then to give the answer. The student may use a calculator to do the actual computation.

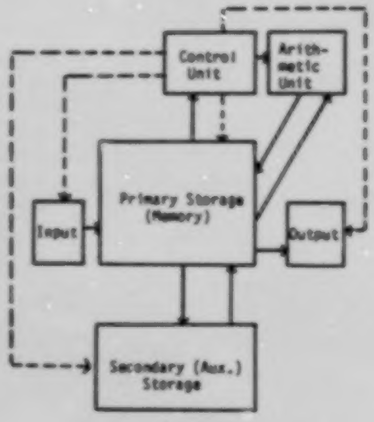
    Steps in solving the problem:

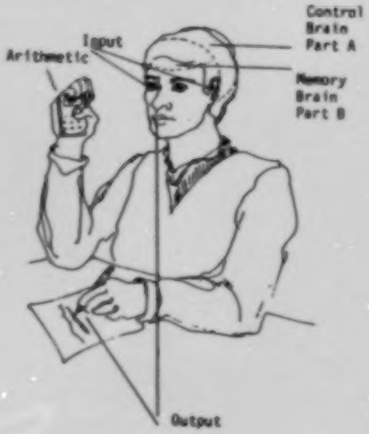
    a) The student hears the two numbers with his/her ears (input unit).
    b) One part of the brain (PART A - Control Unit) transfers the numbers from the ears to another part of the brain (PART B - Memory Unit) where the numbers are remembered.
    c) PART A of the brain (control) then causes the numbers to be copied from PART B (memory) into the calculator (arithmetic unit).
    d) PART A (control) then instructs the calculator (arithmetic) how to perform the calculation.
    e) PART A (control) transfers the answer from the calculator (arithmetic) to PART B (memory).
    f) Now the student is ready to reply to the question. PART A (control) causes the answer to be transferred from PART B (memory) to the mouth (output unit), from which the answer is given.

OBJECTIVES

NOTES AND EXPLANATION

This procedure can be followed through using a human schematic and then it can be compared to the same procedure using a computer schematic.


Control Unit
Arith-metic Unit
Primary Storage (Memory)
Input
Output
Secondary (Aux.) Storage


Control Brain Part A
Input
Arithmetic
Memory Brain Part B
Output

37

Dotted lines represent the interaction (messages) between the Control Unit and the other units; that is, before any device can perform an operation it must receive a command from the control unit.

38

Solid lines represent movement of user instructions or data.

OBJECTIVES

NOTES AND EXPLANATION

It is important to point out that alternate input and output units could have been used. For example, if the question had been written on the board and the answer was to be written on paper, then the input unit would have been the eyes and the output unit would have been the hand.

3. Compare the processing done by a clerk and a calculation done by a student on a calculator to the operation of a computer.

	CLERK	CALCULATOR	COMPUTER
Input	Files placed in "IN" basket	Buttons punched	Instructions
Storage	Information on desk relating to the problem	Number(s) located in memory	Instructions stored in memory
Control	Clerk decides what task is to be performed	Student decides which operation key to press	As directed by the instructions
Process	Clerk performs specific task	Specific operation key punched	Built in function for arithmetic and logic
Output	Finished work in "OUT" basket	Display	Information is transferred to output device

OBJECTIVES	NOTES AND EXPLANATION

2. identify the devices used with each component;

Input Devices	Storage Devices	Output Devices
punch card reader	memory chips (LSI-large scale integrated)	line printer
paper tape readers	magnetic cores	C.R.T. screen
magnetic ink character readers	magnetic disks	card punches
optical code readers	magnetic drums	paper tape punches
magnetic tape readers	magnetic cards	graph plotters
C.R.T.'s (cathode ray tubes)	bubble memory	cash registers
typewriters/teletypes		typewriters/teletypes
touch sensitive screens		magnetic tape
audio response units		audio response units

Note: The above lists are not exhaustive. In theory, a computer system could use any input device from list 1, any storage device from list 2, and any output device from list 3.

3. identify factors affecting the selection of input and output devices;

. location - terminal remote vs. close to C.P.U.
. speed - number of bits/characters/lines/pages per second
. cost - purchase vs. rental decision
. security - "crime proof"
. convenience - ease of data entry

OBJECTIVES	NOTES AND EXPLANATION
	. number of users - batch vs. interactive The above factors are considered to be the most important, but other factors which might be considered include compatibility, capability, previous user experience, maintenance, etc.
4. identify factors affecting the selection of storage devices.	. Usage - primary vs. secondary storage Primary usually has no mechanical movement . Access type - sequential vs. random access Random - data can be requested, located, and retrieved without reference to any other items in storage (e.g., magnetic disk) Sequential - all preceding data must be referenced (e.g., magnetic tape) . Portability - removability of the storage media (disk, cassette, tape, floppy) . Capacity - generally expressed in K bytes (1K = 1024 characters) Choice would be affected by the size of the largest job, the number of programs, and the volume of data. . Cost - weighing the advantages and disadvantages of each storage medium, keeping in mind the above four factors. For example, tape is inexpensive but fast. These factors are considered to be the most important but other factors that might be considered are compatibility, capability, previous user experience, and maintenance.

B. Hardware and Software

OBJECTIVES	NOTES AND EXPLANATION
The student should be able to:	
1. distinguish/define hardware, software, and firmware;	HARDWARE: The physical, electronic, and electromechanical devices such as the central processing unit, printers, card readers, tape drives, and disk drives. SOFTWARE: The programs that control and coordinate the activities of the computer hardware and which direct the processing of data. Software packages are groups of programs that refer to a particular application, e.g., accounting, statistics, sorting. FIRMWARE: The micro programs, that is, the sequencing programs for the control unit of any processor which interpret the external instruction set of a computer. This term is frequently extended to describe any programs residing in ROM. ROM: Read Only Memory, usually programmed by the manufacturer. ROM gives high bit density, non-volatility, and low cost. The contents are fixed and do not store user programs. RAM: Random Access Memory, the read and write memory. This is where programmers typically store data programs during program execution. PROM: Programmable Read Only Memory, which can be programmed directly by the user using a special PROM programmer device. Each memory cell is manufactured with a fuse, and during the programming process the PROM programmer device generates trains of pulses causing selected fuses within the memory to be blown. This causes row/column junctions to be open at these locations; hence the term "burn a PROM." If large quantities are to be manufactured then ROM is used. The disadvantage with ROM and PROM is that the memory content is "fixed." EPROM: Erasable PROM or programmable PROM. A typical EPROM can be erased by exposing it to ultraviolet light for five to ten minutes where all the memory cells are returned to zero or discharged. Once zeroed, memory can then be reprogrammed using a special EPROM programmer device.

OBJECTIVES

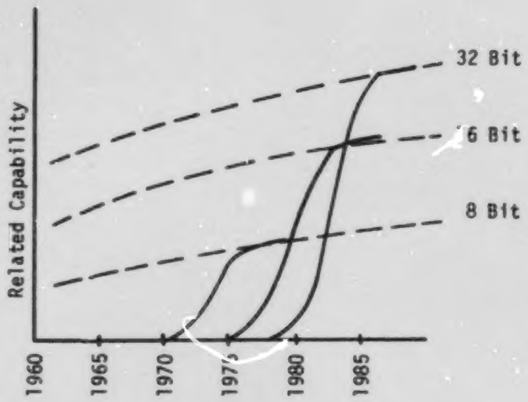
NOTES AND EXPLANATION

EEPROM: Electronically Erasable PROM.

2. discuss micro, mini, and maxi computer systems according to cost, memory, and speed.

The most important differences between each of these computer types is size of memory. In general, "mainframes" or "maxis" are found in major corporations such as banks and insurance companies, large universities, and major government departments. "Minis" would be used by smaller companies such as accounting firms, and "micros" can be found in offices, schools, and even automobiles.

Most industry people have a difficult time classifying these types of machines. The reason for this is that the micro has duplicated the processing capability of the mini in a very short time frame and at a much reduced cost. Additionally they have gone from 8 bit to 16 bit to 32 bit processors in a ten-year period (1972 to 1982).



------ represents minis
——— represents micros

Relative Capability includes processing speed, auxiliary storage, number of input and output devices that can be attached, etc.

For example, in the early 1980s a 16 bit micro can match a 16 bit mini and in some cases exceed it. A similar situation will likely happen with the maxi, so that the distinction between micro, mini, and maxi will become even more blurred in the future.

To distinguish clearly among the three categories, especially minis and maxis, is a difficult task. The following table identifies some characteristics which may aid in the distinction:

	MICROS	MINIS	MAXIS
Date of Introduction	mid 70s	mid 60s	mid 50s
Operating System	etched into memory (ROM)	entered into memory in program form each time the machine is turned on	same as mini
Capacity	4K → 64K memory expandable to 128K plus	up to 2M main memory	up to 16M main memory
Word Length	most 8 bit, some 16 bit words	16 to 32 bit word length with parallel processing	16 to 60 bit word length
Auxiliary Storage	usually cassette or floppy disk	usually on tape or fixed disk	usually on fixed disk or tape
Environment	no special environment required (including hydro)	no special environment required (including hydro)	controlled environment and hydro required
Approximate Price Range	$300 ——→ $50 000	$20 000 ——→ $200 000	$1 000 000 ——→ $5 000 000

C. Stored Programs

OBJECTIVES	NOTES AND EXPLANATION
The student should be able to:	
1. execute a "canned" program given appropriate documentation.	In order to provide the student with quick, hands-on experience with a computer, even before any programming is taught, and to demonstrate the fact that programs can have universal and long-lasting use, it is suggested that some time be devoted to the use of "canned programs" early in the course. If the Manitoba High School Network is being used, some PLUM programs such as BIORC or YORICKC might be attempted. The use of MAPO cards could be introduced. If microcomputers are being used then students should be given the opportunity to run programs which have been stored on cassette or diskette.

UNIT II. PROBLEM SOLVING

OBJECTIVES	NOTES AND EXPLANATION
The student should be able to:	
1. discuss various methods used in solving problems;	When a teacher thinks of "problem solving," he or she probably thinks of methods used in arriving at the solution to a written or oral problem. Such methods might include: a) guessing, b) trial and error, c) pattern searching, d) working backward from the answer (if known), e) following series of instructions (algorithm). These methods might apply to any problem that needs to be solved, but in Computer Science 205, teachers should inform students that although humans can use all of the above techniques and more, the computer can use only one of them - the last one.
2. identify the logical series of steps involved in solving a problem;	One approach to solving a problem involves the following steps: Step 1: Understand the problem and define it completely. Consider the following: a) Determine the output required (create a sample(s)). b) Determine the input. Construct a sample printout of the input and output to give a quick synopsis of the problem. This can prevent conflicts in cases where the output turns out to be quite different from the expectation of the person

OBJECTIVES

NOTES AND EXPLANATION

defining the problem. If a person is not given a sample of the output, he or she should try to provide a sample before writing the instructions.

c) Define the problem completely. Concern yourself with: incomplete problem definition, missing information, ignorance of special cases, extraneous informtion.

In order to analyze a problem, the student must be able to read and understand the problem. The teacher should be aware of the following causes of poor problem-solving techniques:

(i) vocabulary or reading inefficiencies which may cause failure to understand the problem,
(ii) lack of reasoning ability,
(iii) short attention span or lack of interest,
(iv) lack or absence of a systematic attack,
(v) lack of ability to choose essential data.

Step 2: Find the connection between the given data, and the required output. (Devise a plan.)

Consider the following:

a) If you or someone else has solved a similar problem previously, use it as a guide.

b) Can the problem be restated?

c) Use a "top-down" approach.

Design in levels, where a level consists of one or more modules. For example, most programs have a first level which can consist of the module's initialization, process, and termination. Each of these may utilize any number of submodules at the next level. Note that a module is "complete"

OBJECTIVES	NOTES AND EXPLANATION

in that it specifies all the tasks that it wants to perform even though the detail may be specified in a lower module. The number of levels is determined by the complexity of a problem.

d) Don't "re-invent the wheel."

If one of your modules or submodules requires a function which already exists then that module is problably complete.

Step 3: Convert the plan into written instructions.

Step 4: Thoroughly test the sequence of instructions to ensure that the output is correct and complete.

Hand-check the sequence of instructions. The technique is simple. Choose a sample input, then calculate the output as if you were the computer, assuming nothing and using exactly what is written. See that each logical unit performs correctly and that the control sequence through the units is correct. If the solution is too long or too complex to check in its entirety, then check each major section first, and later check the smaller units, assuming that the major sections are correct. When choosing sample input, take special care to include the boundary conditions and other unusual cases. Failure to account for these conditions/cases is one of the most common errors in computer solution to a problem.

3. define the term "algorithm";

ALGORITHM: A finite set of instructions which, if followed, accomplishes a particular task.

An algorithm may or may not require external input. In addition, an algorithm must satisfy the following criteria:

a) Output: At least one quantity is produced.

57

58

OBJECTIVES

NOTES AND EXPLANATION

b) Definitenes: Each instruction must be clear and unambiguous.

c) Finiteness: If the instructions of an algorithm are traced out, then for all cases the algorithm will terminate after a finite number of steps.

d) Effectiveness: Every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite: it must also be feasible.

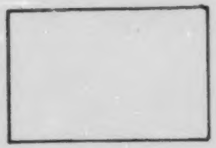
4. define the terms "flowchart" and "pseudo-code";

The pictorial representation of the detailed steps in the solution of a problem is called a flowchart.

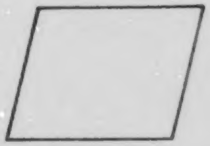
Pseudo-card is an English-type code which allows the program logic to be expressed in a highly readable manner. It often replaces flowcharts and has the advantage that it is close to programming languages (especially COBOL and Pascal). The idea is to allow us to state the logic of the program clearly, ignoring machine and language constraints.

5. correctly identify and to understand the meaning of the flowchart symbols;

FLOWCHART SYMBOLS

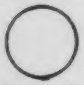


Any processing function--defined operation(s) causing change in value, form, or locations of information

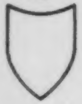


General I/O function--information available for processing (input), or recording of processed information (output)

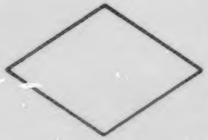
OBJECTIVES	NOTES AND EXPLANATION



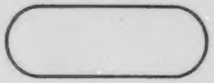
CONNECTOR--exit to, or entry from, another part of chart



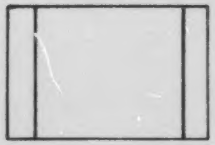
OFFPAGE CONNECTOR--for entry to or exit from a page



A decision or switching-type operation--determines which of a number of alternative paths to follow



A terminal point in a flowchart--start, stop, halt, delay, or interrupt; may show exit from a closed subroutine



One or more named operations or program steps specified in a subroutine or another set of flowcharts

6. recognize three types of logical steps or constructs:

All solutions to problems can be expressed using only three types of logical steps (referred to as constructs). These are:

a) linear (sequence),
b) branching (IF. . . THEN . . . ELSE),
c) looping.

In addition, these can be expanded to include other variations of loops (depending on the richness of the programming language), and case statements (multiple or nested IF . . . THEN . . . ELSE statements).

OBJECTIVES

NOTES AND EXPLANATION

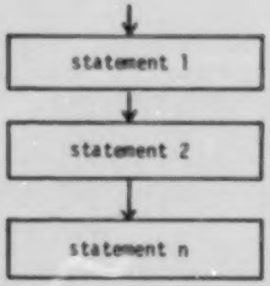
To facilitate the design of a solution to a problem, there are two commonly used representions: flowcharts, and more recently the use of pseudo-code. These may be used independently or in combination with each other. Some students can conceptualize an algorithm more easily using a flowchart, while others find pseudo-code an easier means of expression.

a) linear (sequential),

Linear (Sequence)

A sequence is simply a group of commands performed one after the other in the order in which they appear.

FLOWCHART



PSEUDO-CODE

```
Begin
   statement 1
   statement 2
       .
       .
       .
   statement n
End;
```

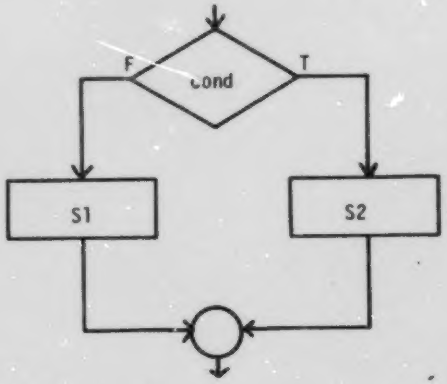
Note: See pages 36 and 37 for specific non-numeric and numeric examples showing linear (sequence).

b) branching (IF . .. THEN
. . . ELSE),

Branching (IF . . . THEN . . . ELSE)

This is a binary decision which will cause one of two sets of statements to be executed. Research has shown that a decision with more than two possibilities is not as clear. A case statement should be considered if more than two possibilities exist.

FLOWCHART



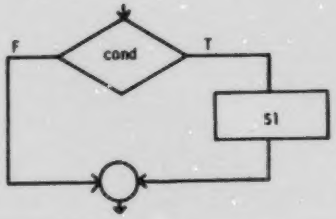
PSEUDO-CODE

```
IF cond THEN
   statement 1  )
   statement 2  )
       .        )  S1
       .        )
       .        )
   statement n  )
ELSE
   statement n+1 )
   statement n+2 )
       .         ) S2
       .         )
       .         )
   statement n+m )
END IF
```

Note: See pages 38 and 39 for specific non-numeric and numeric examples showing branching.

In the above example, SI and S2 are groups of statements in which, conceivably, one of them may contain no statements and look like the following:

FLOWCHART



PSEUDO-CODE

```
IF cond THEN
   statement 1 )
   statement 2 )
       .       )  S1
       .       )
       .       )
   statement n )
END IF
```

OBJECTIVES

c) looping;

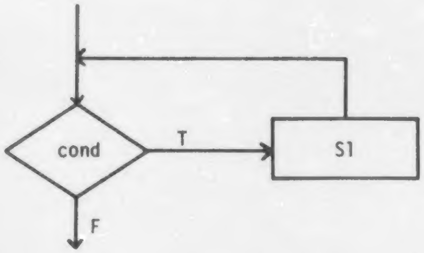
NOTES AND EXPLANATION

Looping

a) WHILE DO

This is a loop which causes a group of statements to be executed while a given condition is true. Hence the loop may be executed zero times, once, or more times.

FLOWCHART



PSEUDO-CODE

```
WHILE cond DO
   statement 1 )
   statement 2 )
       .       ) S1
       .       )
       .       )
   statement n )
ENDWHILE
```

Note: See pages 40 and 41 for specific non-numeric and numeric examples showing looping (WHILE DO).

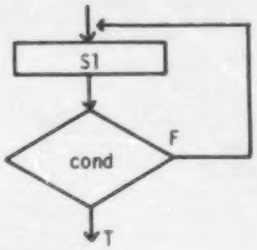
While it has been proven that three constructs are sufficient to write a program, the more expressive a computer language is, the more one can readily accomplish and still maintain control of the design. Following are several other useful constructs, included here for completeness, which should be introduced at a later time.

OBJECTIVES | NOTES AND EXPLANATION

b) REPEAT UNTIL

This is similar to the WHILE loop, but guarantees that the statements will be executed at least once.

FLOWCHART



PSEUDO-CODE

```
REPEAT
   statement 1 )
   statement 2 )
       .       )  S1
       .       )
       .       )
   statement n )
UNTIL cond
```

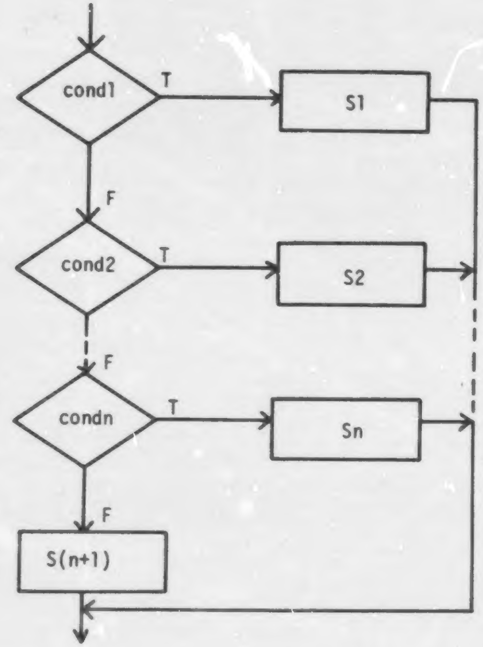
Note: See pages 43 and 44 for specific non-numeric and numeric examples showing looping (REPEAT UNTIL).

Note: The DO loop is the same construct as the REPEAT UNTIL but is implemented using different language syntax.

The Case Statement

This allows one of many possibilities to be chosen and executed. The choices are organized in a uniform manner, and control always enters on top and leaves at the bottom.

FLOWCHART



PSEUDO-CODE

```
CASE
   cond1:S1
   cond2:S2
        .
        .
        .
   condn:Sn
   ELSE:S(n+1)
END CASE
```

NOTE: See pages 45 and 46 for specific non-numeric and numeric examples showing CASE.

OBJECTIVES

7. graphically represent the solution to simple non-numeric and numeric problems using:

   a) flowchart,
   b) pseudo-code;

NOTES AND EXPLANATION

Note that S(n+1) is an error trap for the case where no conditions are satisfied. Under any condition, only one of the sets S1 through S(n+1) is executed.

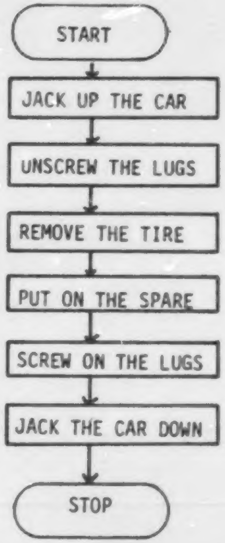
FLOWCHART/PSEUDO-CODE EXAMPLES:

1. Linear

   a) Non-Numeric

   Problem: Draw a flowchart and write the pseudo-code to show the steps in changing a car tire.

FLOWCHART


START
JACK UP THE CAR
UNSCREW THE LUGS
REMOVE THE TIRE
PUT ON THE SPARE
SCREW ON THE LUGS
JACK THE CAR DOWN
STOP

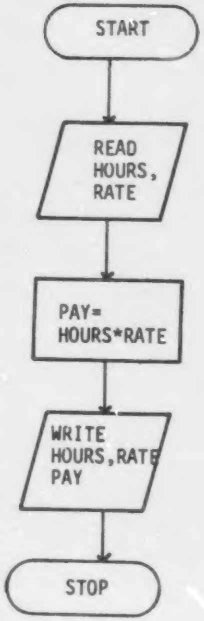
PSEUDO-CODE

```
BEGIN
   Jack up the car
   Unscrew the lugs
   Remove the tire
   Put on the spare
   Screw on the lugs
   Jack the car down
END
```

b) Numeric

Problem: Draw a flowchart and write the pseudo-code to read the hours and rate and then calculate the employee's wages/pay.

FLOWCHART


START
READ HOURS, RATE
PAY= HOURS*RATE
WRITE HOURS,RATE PAY
STOP

PSEUDO-CODE

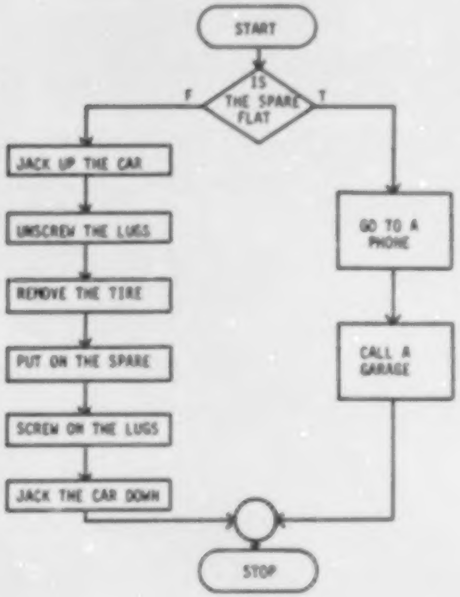
```
BEGIN
   Read Employee Information
   Calculate Pay
   Print Employee Info and Pay
END
```

2. Branching (IF. . .THEN. . .ELSE)

The following examples are modified solutions to the previously stated problems, showing branching. Note: Refer to Appendix C "Conditional Statements" for additional information.

a) Non-Numeric

FLOWCHART

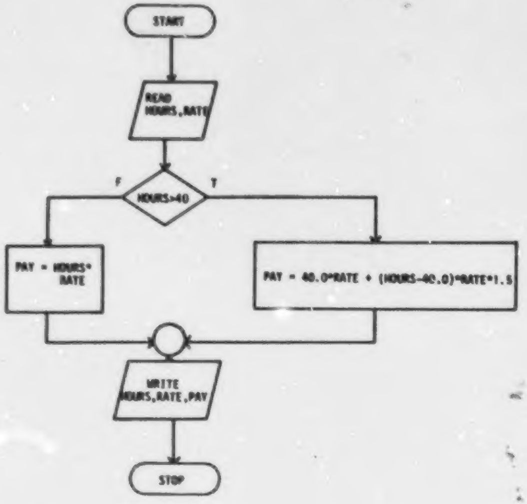

START
IS THE SPARE FLAT
F
T
JACK UP THE CAR
UNSCREW THE LUGS
REMOVE THE TIRE
PUT ON THE SPARE
SCREW ON THE LUGS
JACK THE CAR DOWN
GO TO A PHONE
CALL A GARAGE
STOP

PSEUDO-CODE

```
   Begin
      IF spare flat THEN
      Go to a phone
      Call a garage
   ELSE
      Jack up the car
      Unscrew the lugs
      Remove the tire
      Put on the spare
      Screw on the lugs
      Jack the car down
   END IF
END
```

b) Numeric

FLOWCHART


START
READ HOURS,RATE
HOURS>40
F
T
PAY = HOURS* RATE
PAY = 40.0*RATE + (HOURS-40.0)*RATE*1.5
WRITE HOURS,RATE,PAY
STOP

PSEUDO-CODE

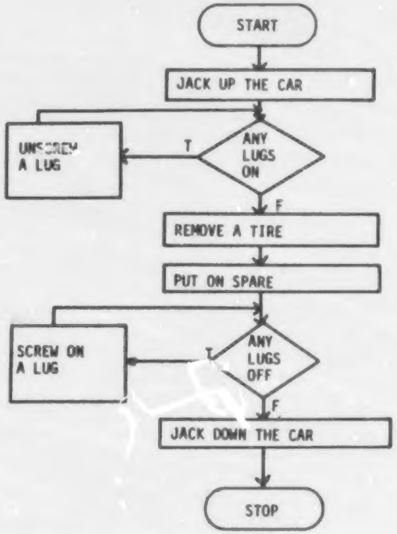
```
BEGIN
   Read Employee Information
   IF Hours > 40 THEN
      Pay = 40.0*Rate+(Hours -
            40.0)*Rate*1.5
   ELSE
      Pay = Hours*Rate
   END IF
END
```

3. Looping (WHILE DO)

The following examples are modified solutions to previously stated problems showing looping (WHILE DO).

a) Non-Numeric

FLOWCHARTING

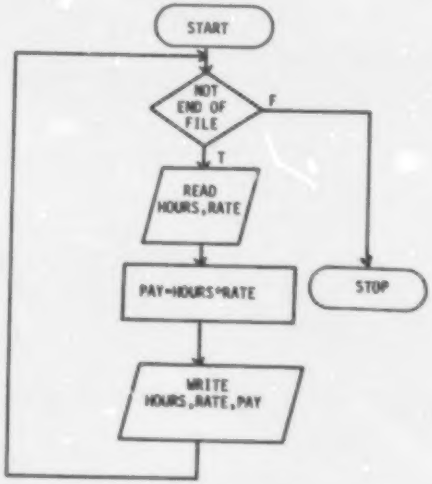

START
JACK UP THE CAR
ANY LUGS ON
T
UNSCREW A LUG
F
REMOVE A TIRE
PUT ON SPARE
ANY LUGS OFF
T
SCREW ON A LUG
F
JACK DOWN THE CAR
STOP

PSEUDO-CODE

```
BEGIN
   Jack up the car
   WHILE any lugs on DO
      Unscrew a lug
   END WHILE
   Remove a tire
   Put on the spare
   WHILE any lugs off DO
      Screw on a lug
   END WHILE
   Jack down the car
END
```

b) Numeric

FLOWCHARTING


START
NOT END OF FILE
F
T
READ HOURS,RATE
PAY=HOURS*RATE
STOP
WRITE HOURS,RATE,PAY

PSEUDO-CODE

```
BEGIN
   WHILE not end of file DO
      Read Hours, Rate
      Pay = Hours * Rate
      Write Hours, Rate, Pay
   END WHILE
END
```

Trace of Flowchart (page 41)

Sample Data Deck			Predicted Pay
a)	40.0	5.00	$200.00
b)	34.0	4.50	$153.00
c)	42.5	6.00	$255.00
d)	EOF		

Trace:

Step	Not EOF	Hours	Rate	Pay
1	T	40.0	5.00	200.00
2	T	34.0	4.50	153.00
3	T	42.5	6.00	255.00
4	F(Stop)			

Output:

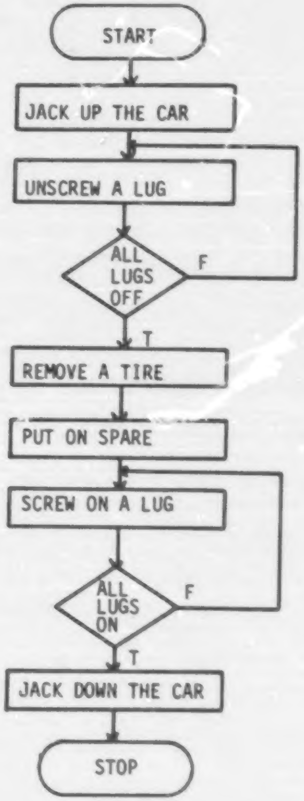
```
40.0        5.00        200.00
34.0        4.50        153.00
42.5        6.00        255.00
```

As the output from the flowchart matches the predicted results, the flowchart must be correct.

4. Looping (REPEAT UNTIL)

a) Non-Numeric

FLOWCHART


START
JACK UP THE CAR
UNSCREW A LUG
ALL LUGS OFF
F
T
REMOVE A TIRE
PUT ON SPARE
SCREW ON A LUG
ALL LUGS ON
F
T
JACK DOWN THE CAR
STOP

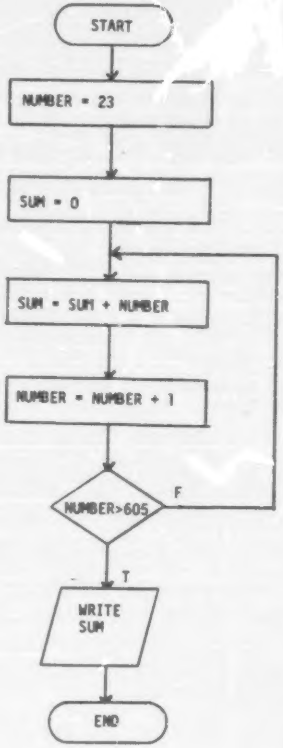
PSEUDO-CODE

```
BEGIN
   Jack up the car
   REPEAT
      Unscrew a lug
   UNTIL all lugs off
   Remove a tire
   Put on spare
   REPEAT
      Screw on a lug
   UNTIL all lugs on
   Jack down the car
   END
```

b) Numeric

Problem: Draw a flowchart and write the pseudo-code to add the integers 23 to 605 inclusive.

FLOWCHART


START
NUMBER = 23
SUM = 0
SUM = SUM + NUMBER
NUMBER = NUMBER + 1
NUMBER>605
F
T
WRITE SUM
END

PSEUDO-CODE

```
BEGIN
   NUMBER = 23
   SUM = 0
   REPEAT
      SUM = SUM + NUMBER
      NUMBER = NUMBER + 1
   UNTIL NUMBER > 605
   WRITE SUM
END
```

OBJECTIVES

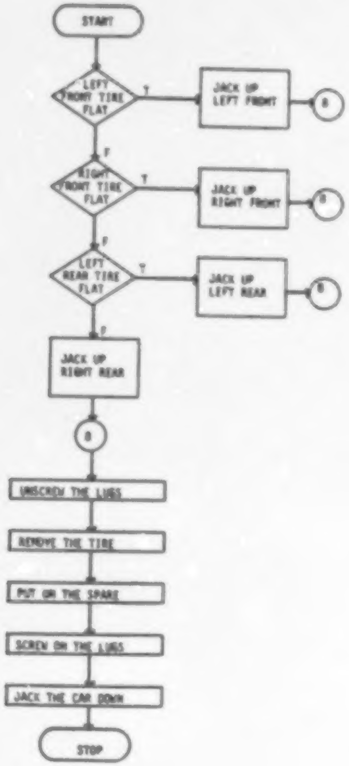
NOTES AND EXPLANATION

5. Case

a) Non-Numeric

Problem: Draw a flowchart and write the pseudo-code for changing a car tire.

FLOWCHART


START
LEFT FRONT TIRE FLAT
T
JACK UP LEFT FRONT
B
F
RIGHT FRONT TIRE FLAT
T
JACK UP RIGHT FRONT
B
F
LEFT REAR TIRE FLAT
T
JACK UP LEFT REAR
B
F
JACK UP RIGHT REAR
B
UNSCREW THE LUGS
REMOVE THE TIRE
PUT ON THE SPARE
SCREW ON THE LUGS
JACK THE CAR DOWN
STOP

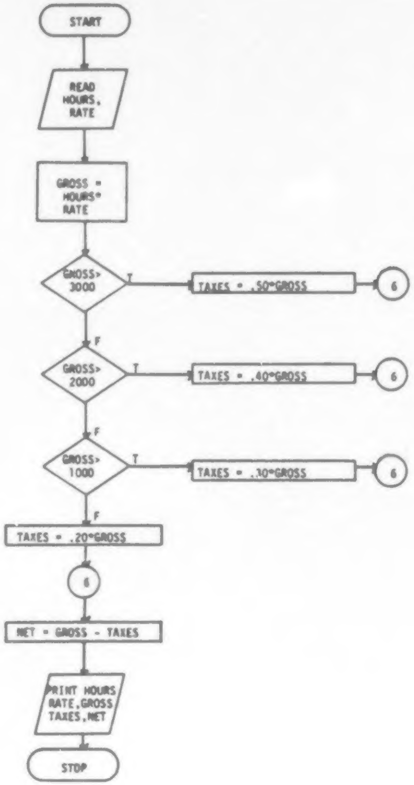
PSEUDO-CODE

```
BEGIN
   CASE
      Left front flat:
         Jack up left front
      Right front flat:
         Jack up right front
      Left rear flat:
         Jack up left rear
      Right rear flat:
         Jack up right rear
   END CASE
   Unscrew lugs
   Remove tire
   Put on spare
   Screw on lugs
   Jack the car down
END
```

b) Numeric

Problem: Draw a flowchart to write the pseudo-code for calculating an employee's net pay given hours, rate, and tax information.

FLOWCHART


START
READ HOURS, RATE
GROSS = HOURS* RATE
GROSS> 3000
T
TAXES = .50*GROSS
6
F
GROSS> 2000
T
TAXES = .40*GROSS
6
F
GROSS> 1000
T
TAXES = .30*GROSS
6
F
TAXES = .20*GROSS
6
NET = GROSS - TAXES
PRINT HOURS RATE,GROSS TAXES,NET
STOP

PSEUDO-CODE

```
BEGIN
   Read Hours, Rate
   Gross = Hours* Rate
   CASE
      Gross  3000 : Taxes = .50* Gross
      Gross  2000 : Taxes = .40* Gross
      Gross  1000 : Taxes = .30* Gross
      Gross  0 : Taxes = .20* Gross
   END CASE
   Net = Gross - Taxes
   Write Hours, Rate, Gross, Taxes, Net
END
```

93

94

OBJECTIVES	NOTES AND EXPLANATION
5. test his/her algorithm.	For numeric problems, the student should set up a table with all variables listed and trace the value of each during execution of the instructions. If looping is involved, continue long enough to determine if the proper pattern is established and if it terminates with the desired values. Refer to page 43.

III. PROGRAMMING

OBJECTIVES	NOTES AND EXPLANATION

The Importance of SPECTRE and SPECTRE MAP

NOTE: The pseudo-machine and pseudo-assembly language sections of the course are to be alloted a maximum of 10% of the available teaching time.

SPECTRE and SPECTRE MAP are a pseudo-machine and a pseudo-assembly language which have been included in the course, more as an aid to the student's understanding of how the computer works on a high-level language program, than as a means of creating machine and assembly language programmers. Knowledge of programming at these lower levels will enhance a student's ability to program with high-level languages. It is important that students understand what the computer does when a program in languages such as FORTRAN, Pascal, COBOL, etc., is submitted (i.e., what occurs during compile time and execution time).

SPECTRE provides a simple but powerful model of information being stored in memory locations, and allows the student to see how the computer distinguishes between data and instructions and then between load data and run data. The close examination of how instructions such as: READ, MULTIPLY, or DIVIDE operate, will give the student a greater appreciation of how these operations are carried out in a high-level language. It is important that these machine-language commands be related to their corresponding high-level commands, whether SPECTRE is taught prior to the introduction of the high-level language or at some time during the teaching of the high-level language (often just before the introduction of subscripted variables).

The most important concept to convey in SPECTRE MAP is that the load phase requires two passes, since the mnemonic opcodes and labels must be translated to numeric operation codes and numeric addresses in order that the computer may operate upon them. This translation process can be related directly to what occurs when a high-level language program is processed. The student should realize that the computer actually works in machine language and this can be

OBJECTIVES

NOTES AND EXPLANATION

emphasized by having the student translate several programs from SPECTRE MAP into SPECTRE.

Remember, the main reason for including these sections in the course is to give the student a better understanding of what the computer is doing with his/her high-level language program, not to create proficient machine and assembly language programmers.

Using the SPECTRE Diskette on a Micro (Apple, PET, TRS-80)

NOTE: One manual suitable for Apple, PET, and TRS-80 is available from the Department.

1. Insert the SPECTRE diskette into drive 1. Type PR#6 and press return. This should give you the menu as follows:

```
**MANITOBA DEPARTMENT OF EDUCATION**
         SPECIFIC PROGRAMS
   PLEASE SELECT ONE OF THESE OPTIONS:

<E> EDIT OR CREATE A SPECTRE FILE
<D> DELETE A SPECTRE FILE
<I> SPECTRE INTERPRETER
<A> SPECTRE ASSEMBLER
<L> LIST SPECTRE PROGRAMS
<Q> QUIT SPECTRE PROGRAM
```

2. Since you want to enter a program you will type E. The computer will respond:

```
ENTER NAME OF FILE TO BE EDITED/CREATED.
??
```

You can type in a name that is up to 8 characters long.

OBJECTIVES	NOTES AND EXPLANATION

3. If there isn't a file already on the diskette with that name the computer will respond:

```
YOU ARE NOW IN EDIT MODE OF FILE
'TEST'
FOR MORE INFORMATION ENTER 'HELP'
FILE 'TEST' IS EMPTY
```

4. You then type:

```
INSERT
EX. INSERT
```

5. The computer will respond with:

```
         < INSERT MODE>
COLUMN: -2------9---13
     1:
```

You type in your SPECTRE program. Remember the first line starts with a 1 and your comment.

6. When you are finished the computer will respond with:

```
<COMMAND MODE>
??
```

You then type END

7. After you type END the computer will respond with:

```
EDITOR END
PRESS <SPACE BAR> TO CONTINUE
```

8. This should give you the menu. You will type I. The computer then responds with:

```
ENTER NAME OF SOURCE FILE: ▨
                        (cursor)
```

OBJECTIVES	NOTES AND EXPLANATION
	9. You type in the name of the file you just created and press RETURN, the computer then responds with: SPECTRE LOAD PHASE STARTED 10. The computer then asks where you want the output. You will type S for screen unless you have a printer hooked up, in which case you can type P for printer.

103

104

A. Pseudo-Machine Language

OBJECTIVES

The student should be able to:

1. describe the characteristics of the five basic parts of a computer as they relate to a pseudo-machine language.

NOTES AND EXPLANATION

Much of the information and explanation in this guide on pseudo-machine and pseudo-assembly languages is either quoted or paraphrased with permission from: Brillinger, P. C., B. L. Ehle, and J. W. Graham. An Introduction to the SPECTRE COMPUTER. Waterloo: Department of Applied Analysis and Computer Science, University of Waterloo, 1968.

a) Memory

The SPECTRE computer has a memory or storage unit which consists of 1000 storage locations called "words." Each word is capable of recording any 10-digit signed integer. The largest integer in any one word is +9999999999 and the smallest is -9999999999. The integer 5 would be stored as +0000000005, whereas the integer -265 would be -0000000265. Zero is either +0000000000 or -0000000000.

These 1000 words each have a name which we refer to as an "address." These addresses are 000, 001, 002, 003, . . ., 997, 998, 999. Consequently we speak of word 003 or word 576, for example. When we speak of "the contents of word 422," we mean the 10-digit signed integer stored in that word.

b) Input

The SPECTRE computer has an input unit which consists of a card reader in a batch system or a keyboard in an interactive system. The card reader reads information from either keypunch cards or MAPO cards. The information may be data or instructions depending on the format of the record.

OBJECTIVES	NOTES AND EXPLANATION

c) Output

The SPECTRE computer has a printer for its output unit. The printer outputs the contents of one "word" per line.

d) Arithmetic/Logic

The arithmetic unit is used to add, subtract, multiply or divide any two integers in the SPECTRE memory. It performs addition using a special extra word of memory called the "accumulator" (AC). One integer is placed in the AC, another is added to it, and the sum is stored in the AC. Subtraction is similar. The AC also handles the comparison of two numbers. Division and multiplication require still another special word called the "multiplier-quotient" (MQ). Both the AC and the MQ are non-addressable words in the SPECTRE memory.

e) Control

The control unit interprets instructions which are stored in memory and causes them to be executed.

Each instruction consists of two parts, an "operation code" and an "address." The operation code is a two-digit integer which instructs the machine to perform some operation such as print, add, subtract, multiply, divide, and stop. The address part of the instruction is three digits and usually refers to the address of the word which is to be printed, added, subtracted, multiplied, or divided. For a complete list of operation codes see Appendix A.

The instruction is stored in a word of SPECTRE memory as follows:

+00000XXWWW

OBJECTIVES

NOTES AND EXPLANATION

The XX is the two-digit operation code. The WWW is the three-digit address. The high-order digits are always zeros to fill out the word. The sign of an instruction is positive.

A typical instruction would be:

+0000031245

This instruction has operation code 31, which means "print," and the 22245 tells the machine what is to be printed. In other words, this instruction means "print the contents of the word whose address is 245." If we wished to print out four words we would require four such instructions, each with the same operation code, but a different address. For example, the sequence of instructions:

+0000031245
+0000031246
+0000031506
+0000031721

would print on the output printer the contents of words 245, 246, 506 and 721 respectively on separate lines, one beneath the other.

Any sequence of instructions which leads to the solution of some problem is called a "program." Since the SPECTRE computer is a "sequential" computer, instructions must be placed in consecutive words of memory. This means that, having executed the instruction found in a given word of memory, the SPECTRE computer normally finds its next instruction in the next word of memory.

The SPECTRE machine is a "digital" computer because it performs its operations with integers consisting of digits. It is "automatic" because the control unit interprets the instruction, one after the other, automatically and without human intervention.

109

110

OBJECTIVES

2. identify the following pseudo-machine language statements:

    a) load statements,
    b) control statements;

NOTES AND EXPLANATION

Load Statements

a) Data

To place the integer +0026217903 into the word 700, code the word address into columns 2, 3, 4 and the integer +0026217903 would be coded from column 6 to column 16 inclusive. Note that the sign is always in column 6. The integer must be filled out with zeros in the high order positions so that it becomes a 10-digit integer. Columns 1 and 5 and columns 17 to 80 are left blank.

This record is referred to as a "load card," and when input, the integer +0026217903 is automatically stored in word 700. Similarly 730 -0000234567 would automatically store the integer -0000234567 into word 730.

b) Instructions

If a problem is required to add the contents of words 700 and 730 and print the results, five instructions must be placed in memory.

These instructions must be loaded into five consecutive words of memory. In order to get these instructions into memory, we must code them into "load cards" as follows:

```
401  +0000010700
402  +0000020730
403  +0000011740
404  +0000031740
405  +0000040000
```

Instructions use the same columns as data load cards.

OBJECTIVES

3. develop programs using the load and control statements;

NOTES AND EXPLANATION

Control Statements

The SPECTRE computer requires two control cards. First, place a card with a "1" coded in column one at the start (front) of the load "deck." This instructs the machine that SPECTRE load cards are following. Second, place a card with a "4" punched in column one and "401" punched in columns 2, 3, 4 at the back of the load deck. This instructs the computer that loading has terminated, and that the instructions should be executed beginning with the one stored in 401.

The complete input deck is coded as follows:

```
 1                                               ┐
  401   +0000020700 ┐                            │
  402   +0000020730 │                            │
  403   +0000011740 }        instructions        │
  404   +0000031740 │                            ├── load cards
  405   +0000040000 ┘                            │
  700   +0026217903 }        data                │
  730   -0000234567 ┘                            │
 4401                                            ┘
```

The first and last records are called "control cards." These are, in a sense, "start loading" and "end loading" buttons on the computer. Instead of using buttons, we code the controls into records so that the computer operation is "automatic."

The following is a sample of a pseudo-machine language program which consists of a single <u>sequence</u> of instructions.

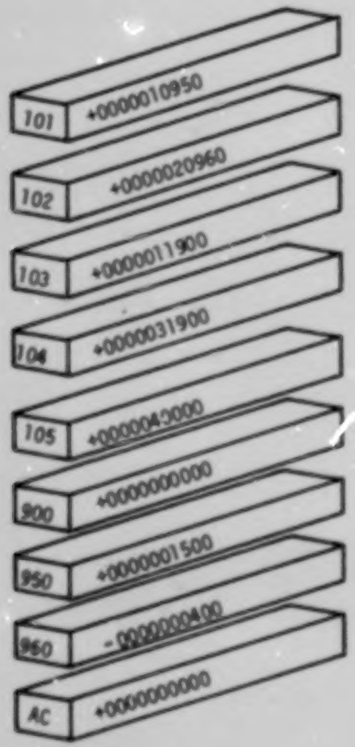
```
1 EXAMPLE #1 CS205             Interpretation of Instructions

 101 +0000010950    a)  clears AC and stores the contents of address 950
                        into AC.
 102 +0000020960    b)  adds the contents of address 960 to contents of AC
 103 +0000011900    c)  stores the contents of AC in address 900
 104 +0000031900    d)  prints the contents of address 900
 105 +0000040000    e)  halts
 950 +0000001500
 960 -0000000400        data
4101
```

NOTE: The contents of the AC cannot be printed directly from the AC.

What is the output?

1500 + (-400) = +0000001100

Go over what happens in memory as the program is loaded and executed using a diagram of memory locations. A suitable example appears on the next page.


101
+0000010950
102
+0000020960
103
+0000011900
104
+0000031900
105
+0000043000
900
+0000000000
950
+0000001500
960
-0000000400
AC
+0000000000

OBJECTIVES	NOTES AND EXPLANATION
4. run a pseudo-machine language program on a computer;	In order to run a pseudo-machine language program on the High School Network, the following cards are required. JOB card USER card SPECTRE. EOR card 1 ⎫ ⎬ load cards (include instructions and data) 4 ⎭ $\underbrace{- - -}_{\text{address of first instruction}}$ E01 card NOTE: Other systems will have different formats for the use of SPECTRE machine language. Refer to the instruction manual for your particular machine.
5. identify and to describe the two phases involved in processing a program: a) load, b) run;	The processing of a program is under direct guidance of the control unit. Knowledge of two registers (special, non-addressable memory locations) is helpful in understanding how the control unit functions. These registers are as follows: a) Instruction Address Register (IAR) This special 3-digit word obtains its first value from the 4 "card" which is the address of the first executable instruction. Example: 4301 loads 301 into the IAR It increments by 1 upon each instruction's execution and in this way stores the address of the next instruction to be executed.

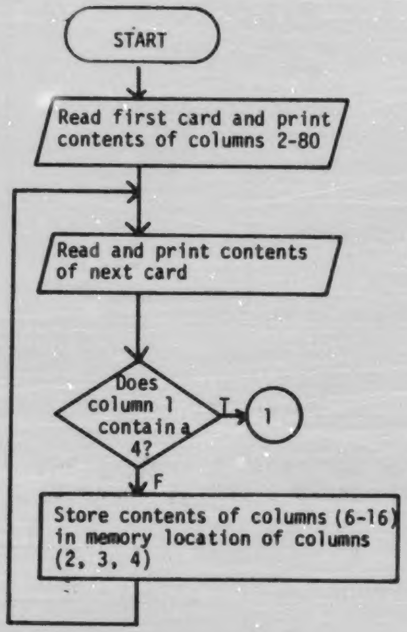
OBJECTIVES	NOTES AND EXPLANATION

b) Instruction Register (IR)

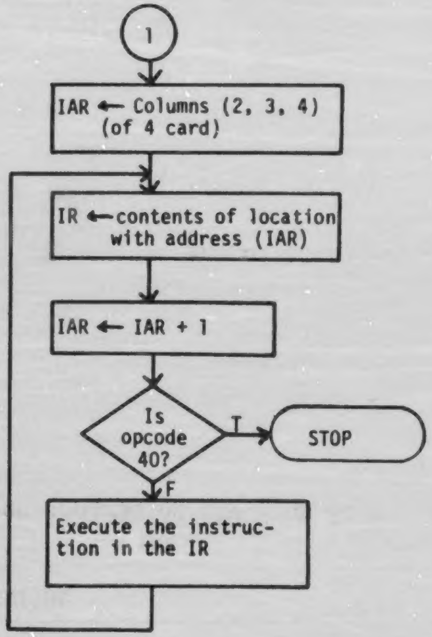
This signed 10-digit "word" stores the instruction currently being executed.

The following flowchart gives a simple overview of the two phases involved in the processing of a pseudo-machine language program. The student will gain a better understanding of how a program is processed in two distinct phases if this flowchart is referred to in relation to several examples. It is particularly helpful for distinguishing between load-time data and run-time data.

Load Time


START
Read first card and print contents of columns 2-80
Read and print contents of next card
Does column 1 contain a 4?
T
1
F
Store contents of columns (6-16) in memory location of columns (2, 3, 4)

Execution or Run Time


1
IAR ← Columns (2, 3, 4) (of 4 card)
IR ←contents of location with address (IAR)
IAR ← IAR + 1
Is opcode 40?
T
STOP
F
Execute the instruction in the IR

121

122

OBJECTIVES

NOTES AND EXPLANATION

6. interpret error diagnostics involving:

   a) load time,
   b) run time;

Load-Time Error

If the SPECTRE control unit detects a load time error (i.e., a syntax error):

. the error message appears on the line before the error;
. execution is prevented.

Run-Time Error

If the SPECTRE control unit detects a run-time error, such as an overflow or invalid operation code, execution of the program automatically ceases and an error "dump" occurs.

This "dump" provides the following information to assist the programmer in "debugging" the program:

. an error message (e.g., invalid opcode);
. the address of the memory location containing the instruction which caused execution to cease;
. a copy of the instruction which caused the error;
. the contents of the AC and MQ at the time execution ceased;
. a listing of the contents of all memory locations containing non-zero integers.

7. write simple programs using input statements and run-time data;

Run-Time Data and Input Statements

In many situations it is desirable to enter new information into the computer during the execution of the instructions, as opposed to entering all the data during the load phase. This application is referred to as run-time data or execution-time data, and is accomplished by means of an input statement (operation code "30").

The following is the original example to add two numbers, but this time the values enter memory during execution time.

8. write programs using the following constructs:

   a) sequence,
   b) looping,
   c) branching;

```
1 ORIGINAL EXAMPLE WITH RUN-DATA CS205
 099 +0000030950
 100 +0000030960
 101 +0000010950
 102 +0000020960
 103 +0000011900
 104 +0000031900
 105 +0000040000
4099
+0000001500
-0000000400
```

To emphasize the difference between run-time data and load-time data one could load and execute these instructions in memory boxes and compare with what occurred in the original load-data program, following the flowchart given in objective number 5.

Writing Program Having the Constructs

a) Sequence

All the examples given have been the straight sequence type. Other examples which could be used are:

(i) subtract or multiply two numbers.
(ii) add, subtract, or multiply three numbers.
(iii) add two numbers, and then multiply the sum by a third number, and print the product.

b) Loop

Before looping can be accomplished, the unconditional transfer instruction (operation code "50") must be introduced. This instruction works in the following manner: If an instruction +0000050444 is encountered, then the IAR is set to 444 and the next instruction is retrieved from address 444. In this way there has been a transfer from the sequential flow of control to the instruction in address 444.

Example:

This program uses a loop to double each of several integers which are greater than zero. Each number is coded on a data record and is printed with its doubled value.

```
1 EXAMPLE WITH UNCONDITIONAL TRANSFER CS205
 450 +0000030700
 451 +0000010700
 452 +0000020700
 453 +0000011800
 454 +0000031700
 455 +0000031800
 456 +0000050450
 457 +0000040000
4450
    data cards
E01
```

When the instruction in address 456 is executed, it sends control back to the instruction in address 450. The instructions in addresses 450 through 456 will be repeated and therefore they form a loop. This loop will be executed as many times as there are data cards (i.e., if there are five data cards, then the loop will be executed five times).

The program does contain an error, however, since the halt instruction (address 457) is never encountered. There is no way out, or "branch" from the loop and an error dump will occur when the input unit attempts to read the E01 card as a signed, ten-digit integer. The error can be corrected by using a conditional transfer instruction which will allow control to branch from the loop to the halt instruction.

c) Branch

In order for branching to occur, one of the conditional transfer instructions (OPERATION CODES: "51," "52," "53," "54") must be used. It is important to note that each of these tests depends upon the contents of the accumulator (AC).

The error in the previous program can be corrected by placing a piece of "dummy" data at the end of the data deck. Since all the given data values are greater than zero, a final data record of +0000000000 and a conditional transfer, "54," will allow the required job to be performed. The program would be:

```
1 EXAMPLE WITH CONDITIONAL TRANSFER CS205
 450 +0000030700
 451 +0000010700
 452 +0000054458
 453 +0000020700
 454 +0000011800
 455 +0000031700
 456 +0000031800
 457 +0000050450
 458 +0000040000
4450
data cards
+0000000000
E01
```

This program will execute in the same fashion as the previous example, except this time the dummy data is encountered and tested in the AC, and the "54" test is true. Therefore, control is transferred to the instruction in address 458 and execution will halt without an error.

OBJECTIVES	NOTES AND EXPLANATION
	Another example of branching being used is a program which reads in several values greater than zero and tests to determine whether each number is even or odd. If the number is even, +0000000002 is printed. If the number is odd, +0000000001 is printed.

B. Pseudo-Assembly Language

OBJECTIVES	NOTES AND EXPLANATION
The student should be able to:	
1. identify and use the necessary control commands for a pseudo-assembly language;	Commence Loading The first record of SPECTRE MAP program has a 5 in column 1, and the programmer's identification in the remainder of the record. This record tells the computer that the following records are written in SPECTRE MAP and an assembly must take place. Halt Loading and Commence Executing The last record in a program would have END in the OPCODE field and the symbolic address of the first instruction to be executed in the ADDRESS field. This corresponds identically to the "4 card" in a SPECTRE program.
2. run a pseudo-assembly language program on a computer;	In order to run a SPECTRE MAP program the students must know the format of the instruction. On each SPECTRE load record containing an instruction, there are three important pieces of information. Columns 2, 3, 4 contain the address of the word or location where the instruction is to be stored. Columns 12, 13 contain the operation code for the instruction. Columns 14, 15, 16 contain the address part of the instruction. In SPECTRE MAP (MAP for short), symbols rather than digits are used for the location, operation code, and address; these are referred to as the LABEL, OPCODE, and ADDRESS. The computer automatically translates these symbolic instructions to their numerical equivalents. A special algorithm is used to do this translation, which is referred to as an "assembler." A MAP program to read two integers, subtract the first from the second, and print the result, would be:

OBJECTIVES

NOTES AND EXPLANATION

```
          A
     O     D
L     P     D
 A     C     R
  B     O     E
   E     D     S
    L     E     S
```

```
START  INP NUM1
       INP NUM2
       CLA NUM2
       SUB NUM1
       STO DIFF
       OUT DIFF
       STP
```

The first instruction has been assigned the symbolic location START. The operation codes are three-letter symbols: See Appendix A for a complete listing of the mnemonic opcodes. The addresses are given symbolic names NUM1, NUM2 and DIFF.

The assembler algorithm would cause the first instruction to be placed in 000. Each subsequent instruction would be assigned to 001, 002, 003, etc. The addresses A, B, C could be assigned anywhere else in memory. In order to have control over their assignment, we always place a "dictionary" of these symbolic addresses immediately following the last instruction.

```
START  INP NUM1
       INP NUM2
       CLA NUM2
       SUB NUM1
       STO DIFF
       OUT DIFF
       STP
```

OBJECTIVES	NOTES AND EXPLANATION

```
NUM1 ⎫
NUM2 ⎬      Dictionary
DIFF ⎭
```

The dictionary causes the assembler to save one memory location (word) for each symbol listed in the LABEL "field." These locations follow in sequence immediately after the instruction locations.

In the above example the actual instructions are stored in words 000 to 006 inclusive. Then NUM1 is assigned to word 007, NUM2 to word 008 and DIFF to word 009.

The MAP program is constructed with one instruction per card or one instruction per line (micros) according to the following format:

LABEL	Columns 2, 3, 4, 5, 6, 7
OPCODE	Columns 9, 10, 11
ADDRESS	Columns 13, 14, 15, 16, 17, 18

Note that the LABEL and ADDRESS symbols can have up to six letters in them. They can also have the digits 0 to 9 with the restriction that they must begin with a letter. The following are legal symbols:

MARY
CHAR22
X

The following are not acceptable symbols:

22 CHAR
A*B
STANLEY

137

138

OBJECTIVES

3. translate a pseudo-assembly language to a pseudo-machine language;

NOTES AND EXPLANATION

The complete program for our example would appear as follows:

```
5 PETER PLUM                          ⎫
 START  INP NUM1                      ⎪
        INP NUM2                      ⎪
        CLA NUM2                      ⎪
        SUB NUM1                      ⎪
        STO DIFF                      ⎪
        OUT DIFF                      ⎬   Load Records
        STP                           ⎪
                                      ⎪
NUM1                                  ⎪
NUM2                                  ⎪
DIFF                                  ⎪
                                      ⎪
        END START                     ⎭

+1234567890                           ⎫
                                      ⎬   Data Records
+5555555555                           ⎭
```

Note that the data records are exactly the same as in the SPECTRE. Only the load records are different.

At load time, the load cards are read into the computer and are translated using the assembler algorithm, and the assembled instructions are stored in memory. Then execution of the program takes place.

The previous program would be translated by the assembler as follows:

OBJECTIVES

NOTES AND EXPLANATION

Location	Instruction
000	+0000030007
001	+0000030008
002	+0000010008
003	+0000021007
004	+0000011009
005	+0000031009
006	+0000040000

and execution would begin at 000.

Some Rules

The example illustrates a number of points about SPECTRE MAP and it is probably worthwhile to summarize these in point form as the following set of rules:

a) The OPCODE is always written as a three-digit mnemonic symbol. The OPCODE is left blank in the dictionary. (See Appendix A for basic SPECTRE instructions.)
b) Every load record in SPECTRE MAP has an entry in the OPCODE field except for the 5 card and the dictionary records.
c) The entry in the OPCODE field is either a symbol which stands for an actual operation code (CLA, ADD, etc.), an instruction to the assembler (END), or blank in the dictionary section.
d) Every symbolic address which is used must appear once, and only once, in the LABEL field of the program. In our example, START, NUM1, NUM2, and DIFF were the only symbolic addresses used and they appear to the left of the OPCODE field. Each of these symbolic addresses should appear at least once in the ADDRESS field. In the example, START appears only once, but NUM1, NUM2, and DIFF each appear twice.
e) In the dictionary section, the ADDRESS is always blank, but the LABEL is never blank; it always contains a symbolic address.

OBJECTIVES	NOTES AND EXPLANATION

4. identify and use literals at load time;

Constants in SPECTRE MAP

Constants can be read into SPECTRE memory at run time, using the INP instruction with data cards. It is also possible, however, to read constants into memory at load time using "literals."

Suppose we wish to add together the integers +0000044444 and -0000033333, print the result, and stop. The program could be written as follows:

```
START  CLA =+0000044444
       ADD =-0000033333
       STO SUM
       OUT SUM
       STP

SUM

       END START
```

Note that in the first two instructions there are literals in the ADDRESS fields, rather than symbols as before. A literal consists of a "=" sign followed by a signed 10-digit integer. The assembler treats the ADDRESS field literally in this case, rather than symbolically. This means that the first instruction will cause the constant +0000044444 to be copied into the AC. The second instruction then causes the constant -0000033333 to be added to the accumulator.

The assembler actually creates a constant for each literal used and stores these constants in SPECTRE memory in what is referred to as the "literal pool." The literal pool uses words beginning immediately following the last word used by the dictionary.

OBJECTIVES	NOTES AND EXPLANATION

In the example, the instructions would be stored in words 000 to 004 inclusive. The dictionary has only one symbol SUM, and this is assigned to word 005. The literal pool requires two words, one for each literal, and words 006 and 007 are used. The assembled program appears as follows:

Location	Instruction or Literal	
000	+0000010006	
001	+0000020007	
002	+0000011005	
003	+0000031005	
004	+0000040000	
005	+0000000000	(saved for SUM)
006	+0000044444	
007	-0000033333	

The first instruction causes the contents of word 006 to be copied into the AC; but word 006 has stored in it the constant +0000044444. Hence, the required effect is achieved.

It is not necessary to write the leading zeros in literals. Also, if the sign is omitted, the literal will be assumed positive. Thus, the above program could be written more conveniently as follows:

```
START  CLA =+44444
       ADD =-33333
       STO SUM
       OUT SUM
       STP

SUM

       END START
```

OBJECTIVES	NOTES AND EXPLANATION
	Literals can be used only in the ADDRESS field of instructions. They cannot be used in the dictionary entries or in the END card.
5. describe the operation of the assembler (i.e., load time has two passes).	Any program which is processed by the SPECTRE MAP assembler goes through two phases just like a program that is run using SPECTRE. The difference is that the load phase consists of two passes as described below.

LOAD TIME

Pass I

The assembler creates a "symbol table" which indicates the memory locations assigned to the various symbolic labels which appear in the LABEL field of the MAP load records. It also creates a table of literals whose addresses cannot be assigned until the end of Pass I, since the assembler does not know how many memory locations are required for the instructions and dictionary until it encounters the "END" record.

In order to determine which memory locations are to be assigned to the labels used in the program, the assembler uses a "location counter" (LC). The value of the LC is initialized at 000 and is incremented by 001 as each new load record is examined. As soon as the assembler reads a load record, it saves the contents of the record on an auxiliary storage unit so that it may examine the contents of the card again during Pass II of load time.

After all the labels in the dictionary have been assigned locations, the literals are assigned consecutive locations by the LC in the order in which they appear in the program. By the time Pass I is complete, the symbol table and literal pool will have been filled.

For the previous example, the contents of the Symbol Table and Literal Pool would be:

OBJECTIVES | NOTES AND EXPLANATION

SYMBOL TABLE

LABEL	LOCATION
START	000
SUM	005

LITERAL POOL

LITERAL	LOCATION
+0000044444	006
-0000033333	007

Pass II

All the instructions stored in the auxiliary storage device, labels in the symbol table, and literals in the literal pool are stored in the allotted memory locations with the instructions starting at address 000. Also, the symbolic MAP opcodes are translated to numeric SPECTRE operation codes before storage. By the end of Pass II, the MAP program will be the same as a SPECTRE program which has gone through the load phase.

EXECUTION-TIME

Since a loaded MAP program is identical to a loaded SPECTRE program, its execution-time will match that of a SPECTRE program.

149

150

C. High-Level Language

OBJECTIVES	NOTES AND EXPLANATION
The student should be able to:	
1. draw analogies between computer programs and everyday tasks;	Before a program is introduced to the students, the term "program" should be defined. PROGRAM: A sequence of instructions executed by a computer Other sequences of instructions similar to a program include, for example, recipes, musical scores, knitting patterns, and model airplane assembly plans. The teacher should state the features of a program and compare them to one of the above sequences of instructions, as follows: a) instructions are sequentially executed; b) the processing has a result (printed output or cake or sweater); c) certain values or objects are operated on (program works with data; recipe works with flour, sugar, etc.); d) sometimes a decision is required (what data to use; how long to cook something or whether to use raw or processed ingredients); e) sometimes an instruction or group of instructions is carried out more than once (e.g., repetition in a selection of music; repetition in a knitting pattern).
2. identify and use the fundamental statements for: a) documentation, b) assignment, c) output, d) input, e) termination;	DOCUMENTATION Completing a program involves more than finding a series of instructions that work and running those instructions on a computer: proper documentation is required. . Poor documentation is a concern to a business person purchasing software. . Preparing the same program with and without documentation should illustrate the clarity of the documented version.

OBJECTIVES

NOTES AND EXPLANATION

Documentation is a package of the different aspects of the program, including:

a) program description--a summary of the problem and what the program can do;
b) logic aids such as a flowchart or pseudo-code--outline how the problem was solved, and could include a list of test values (NOTE: Program description and logic aids are not necessarily part of the program listing.);
c) REMS or COMMENTS--including:
  - author,
  - what the program is supposed to do,
  - question number or program name,
  - list of variables and arrays used and what they represent,
  - certain comments throughout the program to explain sections that could be misunderstood.

  NOTES:
  1. Encourage the use of comments or remarks from the first day and insist that students use them. If students realize that comments are needed from the start, it should become second nature as they complete more programs.
  2. Comments should be short and neatly organized.
     - Throughout the high-level language section, students should be shown where statements or routines could be explained or clarified using short comments.
     - Possibly give the students undocumented programs and have the students document them.

d) final cosmetic work, including (for example):
  - having key parts of the program stand out clearly,
  - refraining from using multiple statements on the same line unless memory is extremely limited,
  - aligning "equal value" statements and indenting sub-statements,

  Example 1:

```
10 FOR X=1 to 20
20     READ M$(X),A,B,C,D
30 NEXT X
```

```
Example 2:  IF (X.GT.Y) THEN
               S=S+1
               T=T+10
            ELSE
               M=M+1
            END IF
```

- . indenting nested loops,
- . using meaningful variable names to help eliminate some remarks or comments (this may be difficult to do if BASIC is used),
- . making the program attractive.

ASSIGNMENT

Assignment may be achieved by:

a) initialization in a statement (discussed in this section),

```
Example:  X=5
          Y=X**2
```

b) use of a READ or INPUT statement (discussed in the I/O section).

An assignment statement is one in which a variable is assigned a value.

VARIABLE: The name given to a memory location

An assignment statement accomplishes two things:

- . it reserves a word in memory;
- . it assigns a value to that word.

Depending upon the language and the computer used, maximum variable size will have to be discussed.

OBJECTIVES	NOTES AND EXPLANATION

Any expression can appear to the right of "=". An expression may consist of:

. a constant,
. a defined variable,
. a combination of variables and constants with operation symbols.

Examples:

```
PI=3.14
RAD=4
VOL=PI*RAD**2
```

NOTES:

1. Emphasize that the use of "=" or ":=" does not actually mean "equals" in the mathematical sense but rather means "is assigned the value of."
2. Only variables which are receiving a value can appear to the left of "=".
3. The variables are not printed; the values represented by the variables are.

Suggestions:

1. Prior to using variables, simple programs such as the ones below could be used and their limitations discussed.

   Examples:

   ```
   PRINT 2+2
   PRINT "TOTAL"
   ```

2. If you do not wish to discuss variable types when starting to program in FORTRAN, the IMPLICIT declaration could be used to declare all variables one type.

3. Good programming technique specifies the variable names under TYPE (if the language allows it).

OBJECTIVES	NOTES AND EXPLANATION

OUTPUT

The purpose of creating a program is to create some type of output; hence the use of the words PRINT, WRITE, DISPLAY.

Output should not be a difficult concept to present. Format-free output will have to be introduced in a general way at the start (i.e., after the students see a copy of output, discuss the format-free output generated by your system). If using BASIC, the use of "," (comma), and ";" (semicolon) to display output could be introduced quite early.

Emphasize:

Print a useful amount of output (number of variables), not too little, with important values absent, and not too much, where useful results are lost in a jungle of printout.

INPUT

A second type of assignment is accomplished by having data read in at execution time.

Suggestions:

1. Without the use of the READ or INPUT statement, any program will do the same thing each time it is run. Show students examples of both types (i.e., with and without a READ) and point out the versatility of the READ statement in a program.

2. Data format will have to be discussed whether using cards or working interactively. The ddata format depends largely on the language and system used.

OBJECTIVES	NOTES AND EXPLANATION
	Emphasize: . that the variables are not read, but the values to be stored in the variables are. When using a loop, each time a READ statement is encountered, a new card is read or the next set of data is used from DATA statement. TERMINATION Encourage the use of termination statements when programming even though they may be optional. In FORTRAN: STOP terminates the processing (i.e., the CPU stops processing); END signals the physical end of the program and/or sub program to the compiler. In BASIC: STOP is available in some versions; END is the last statement in a program, and is used to cease execution. In Pascal: END may appear more than once in a program; and, matched with the BEGIN statement, acts as a bracket around each instruction sequence. Suggestion: Compare the STOP and END to the 4-card and 40000 of SPECTRE.
3. describe and show the application of the conditional statement;	Define conditional statements as those which may occur under certain exactly defined conditions. Contrast these statements with those which are unconditional, such as the assignment and GO TO statements.

OBJECTIVES	NOTES AND EXFLANATION

Refer back to the conditional statements in the SPECTRE section and remind students of the actions they cause. The teacher should show the assembly code necessary to translate an IF statement to assembler.

Example: IF A = 12 PRINT A translates to:

```
CLA A
SUB=12
TNZ GO
OUT A
```

Explain that conditionals can cause the execution of blocks of code according to the outcome of the conditions set by the statement (i.e., conditionals can cause one set of actions to take place if the condition(s) set are true or another set of actions if the condition(s) are false).

Give several real-life logic examples of conditional statements. For example, "If it is raining out, put on a raincoat; otherwise put on a jacket." See Appendix C, "Conditional Statements."

The IF statements should be introduced as the conditional statement with many forms and applications using examples such as the following:
(NOTE: Only one form of the IF is shown in this section as it is very similar in form in BASIC, FORTRAN, and Pascal).

a) The IF statement only performs its operation under certain conditions as set out in the statement:

```
IF A=12 PRINT A
```

This statement will only cause printing of variable A when A has the value of 12; if it does not have this value the print will be ignored.

b) The IF statement can be used to cease operation of a program or subroutine:

BEST COPY AVAILABLE

OBJECTIVES	NOTES AND EXPLANATION

```
IF B=0 GO TO ENDPGN
```

This statement will cause the program to transfer control to the end of the program if B has the value of zero.

c) The IF statement can be used to branch to other parts of a program:

```
IF A$="YES" GOSUB TAXCAL
```

This statement causes execution to transfer to a subprogram if A$ has the value YES.

d) The IF statement may use the ELSE condition which can cause operations to occur if the condition is not true:

```
IF D > 0 THEN
   ABD=D
ELSE
   ABD=-D
```

This statement will assign the value of D to ABD if D is greater than zero or else it will assign the value of -D to ABD if D is less than zero. Notice the use of THEN to help clarify the two choices with the ELSE. This has the effect of the Absolute Value Function.

e) The use of the IF statement should be well chosen to handle all appropriate cases. A clear logical layout of the program with a flowchart will aid in the wise choice of statements.

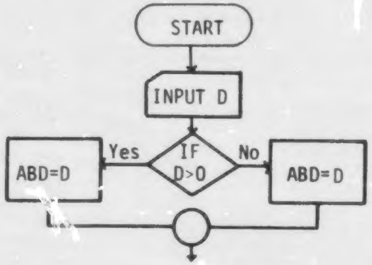
Example: A sample flowchart for the Absolute Value program identified above follows:

165

166

OBJECTIVES

NOTES AND EXPLANATION



f) The IF statement may have multiple conditions assigned by use of the AND, OR, and other keywords:

```
IF F > 4 AND F < 12 PRINT F
```

This statement will only print F if both conditions are met (value of F is between 4 and 12).

```
IF F > 4 OR F < 12 PRINT F
```

This statement will print F if either condition is met (value of F is greater than 4 or less than 12).

Example of IF...THEN...ELSE simulation

a)
```
10 REM PAYROLL PROBLEM
15 REM
20 REM H=HOURS; R=PAY RATE; P=PAY
25 REM
```

OBJECTIVES

NOTES AND EXPLANATION

```
30 PRINT"ENTER HOURS AND RATE"
35 INPUT H,R
40 IF H > 40 THEN 60
45 REM FALSE
50     P=H*R
55     GOTO 70
60 REM TRUE RANGE
65     P = (40*R) + (H-40) *R * 1.5
70 REM ENDIF
80 PRINT H, R, P
85 END
```

or lines 40 - 70 could be

```
b)  40 IF H > 40 THEN 50
    45 GOTO 60
    50     P= 40 * R + (H - 40) * 1.5 * R
    55     GOTO 70
    60 REM ELSE
    65     P = H * R
    70 REM ENDIF
    80 PRINT H, R, P
    85 END
```

Advantages of a) above:

- The false and true ranges are clearly delineated with REMS and there is one less GOTO.
- This is better used with a flowchart.

Advantages of b) above:

- The proper IF...THEN...ELSE... order (i.e., true range preceding false range) is used.
- This is better used with a pseudo-code.

OBJECTIVES	NOTES AND EXPLANATION
	Check your language manual or reference text for specifics on the logical structures and extensions permitted for the conditional statements.
4. identify and use various types of looping techniques;	Definitions LOOP: A series of statements which repeat under certain defined conditions LOOP CONDITION: A condition which must be met in order for a loop to continue its repetition LOOP COUNTER: A variable chosen to count the number of times that a loop has been executed LOOP EXIT: A point in the syntax where execution of the loop may cease and control is transferred to the end of the loop (often the next executable statement after the loop) Explain the importance of well-designed loops which permit repetition as many times as circumstances require Types of Looping: a) Fixed number of repetitions . Often used when an operation requires a fixed number of operations which will not change. Example: Input and add 10 numbers.

OBJECTIVES	NOTES AND EXPLANATION

```
FORTRAN                BASIC                  Pascal (CONST Maxval = 10;)

    TOT = 0            05  TOT = 0            TOTAL :=0;
    DO 25 K=1, 10      10  FOR K = 1 TO 10    FOR COUNT :=1 TO MAXVAL DO BEGIN
       READ, A         20     INPUT A           READLN(NUM);
       TOT = TOT + A   30     TOT = TOT + A     TOTAL :=TOTAL + NUM
25  CONTINUE           40  NEXT K             END.
```

b) Variable number of repetitions

. Found often in programs written to handle many different situations.

Example: Input and add a variable number of values.

```
FORTRAN                BASIC                  Pascal

    TOT = 0            05  TOT = 0            TOTAL :=0;
    READ, N            10  FOR K = 1 TO N     FOR COUNT :=1 TO MAXVAL DO BEGIN
    DO 25 K=1, N       30     INPUT A           READLN(NUM);
       READ, A         40     TOT = TOT + A     TOTAL :=TOTAL + NUM
       TOT = TOT + A   50  NEXT K             END.
25  CONTINUE
```

c) Repetition until a condition is met

. Using repetition to approximate values is common practice and the value is tested after successive approximations until it does not change. This method is often used when a KEY or FLAG is set to halt operation.

Example: Input and add values until a zero is encountered.

FORTRAN

```
   TOT = 0
 5 READ, A
     IF (A .EQ. 0) GO TO 30
     TOT = TOT + A
30 CONTINUE
   PRINT, TOT
```

BASIC

```
05  TOT = 0
10  INPUT A
20     IF A = 0 THEN GO TO 60
30     TOT = TOT + A
40  GO TO 10
50  PRINT TOT
```

. In Pascal the GOTO statement is rarely used, hence the following program:

```
NUMTOT :=0;
READLN(NUM);
WHILE (NUM < > 0) DO BEGIN
    TOTAL :=TOTAL + NUM;
    READLN(NUM)
END.
```

Note that the statements dependent on the loop decision are readily identified. This is certainly not true using the GOTO to create loops.

Example of entrance-controlled and exit-controlled loops:

a) Entrance-controlled loop (similar to a WHILE DO)

- generally used more than the exit-controlled loop;
- unless your version of BASIC has an actual WHILE DO or REPEAT UNTIL it is difficult to do a true simulation.

OBJECTIVES | NOTES AND EXPLANATION

```
10 REM PAYROLL PROGRAM
15 REM EXIT WITH DUMMY DATA=0
20 REM H=HOURS;  R=RATE OF PAY; P=PAY
30 INPUT "ENTER HOURS, RATE ": H,R
35 REM
40 REM BEGIN LOOP (ENTRANCE -CONTROLLED)
45 REM
50    IF H=0 THEN 75
55        P=H * R
60        PRINT H, R, P
65        INPUT "ENTER HOURS, RATE "; H,R
70        GOTO 40
75 REM ENDLOOP
80 END
```

b) <u>Exit-controlled loop</u> (similar to REPEAT UNTIL)

```
10 REM SUM INTEGERS UNTIL SUM > 300
15 REM
20 REM VARIABLES S=SUM: N=INTEGER
25 REM
30 S=0
35 N=1
40 REM BEGIN LOOP (EXIT CONTROLLED)
45     S=S+N
50     N=N+1
55 IF S < = 300 THEN 40
60 REM ENDLOOP
70 PRINT "SUM= "; S
76 END
```

d) Infinite loops

Usually a result of an error in syntax or logic where there is no exit provided or else the condition for exit is not met.

FORTRAN

```
 2  READ, A
       IF (A .GT. 4 .AND. A .LT. 3) GO TO 25
       PRINT, A
       GO TO 2
25  CONTINUE
```

BASIC

```
10  INPUT A
20    IF A > 4 AND A < 3 GOTO 60
30    PRINT A
40  GOTO 10
60  PAUSE
```

Pascal

```
BEGIN
  READLN(NUM);
  WHILE NUM > 4 AND NUM < 3 DO BEGIN
     WRITELN(NUM);
     READLN(NUM)
  END.
END.
```

All of the above programs require a number to be greater than 4 and less than 3 at the same time, which will never happen. All three are infinite loops.

e) Nesting loops

Loops may be placed inside each other to produce repetitions of the repetitions which is very common in sorting. Ensure that the nesting of these loops does not violate rules for the language being used.

Example: The following example will add 10 numbers 5 times.

FORTRAN

```
    DO 40 I=1, 5
       TOTAL = 0
       DO 25 L=1, 10
          READ, A
          TOTAL=TOTAL + A
25     CONTINUE
       PRINT, TOTAL
40  CONTINUE
```

BASIC

```
10  FOR I=1 TO 5
20    TOTAL=0
30    FOR L=1 TO 10
40       INPUT A
50       TOTAL = TOTAL + A
60    NEXT L
70    PRINT TOTAL
80  NEXT I
```

Pascal

```
CONST   MAX1 = 5, ZERO = 0;
        MAX2 = 10;
BEGIN  (*1*)
   FOR I := 1 TO MAX1 DO BEGIN (*2*)
     TOTAL:= ZERO;
     FOR L:= 1 TO MAX2 DO BEGIN (*3*)
       READLN(NUM);
       TOTAL := TOTAL + NUM;
     END  (*3*)
     WRITELN(TOTAL);
   END  (*2*)
END. (*1*)
```

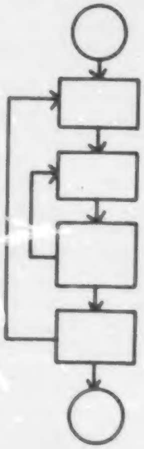
The comments following each BEGIN and END ((*1*)) are used to identify each segment.

OBJECTIVES

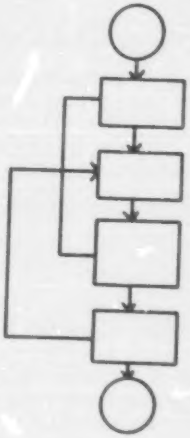
NOTES AND EXPLANATION

Also, refer to the diagrams of nested loops (below).

PROPER NESTING



IMPROPER NESTING



OBJECTIVES	NOTES AND EXPLANATION
5. identify and use formated input and output;	Formated input and output will depend upon the system being used (batch or interactive) and the language being used (BASIC, FORTRAN, etc.) in the school. In some languages and on some machines (APPLE, PET, TRS-80, etc.) different dialects may have different features. Different I/O devices have different key words depending on the system. Teachers should refer to the reference manual for their machine for correct I/O features.

The following statements should be taught for numeric values and for alphabetic characters (strings):

BASIC	FORTRAN	Pascal
INPUT	READ	WRITELN
DATA	PRINT	READLN
READ	FORMAT	WRITE
INPUT#	WRITE	READ
PRINT		
PRINT@ position, item list		
PRINT TAB		
PRINT USING string; item list		
PRINT #		

Stress:

. rules and regulations for field specifiers,
. declaration of variable types (integer, real),
. carriage controls and horizontal spacing,
. rounding, truncation in field lengths,
. string input and output.

Graphics could be used to emphasize proper vertical and horizontal positioning techniques for output. If an interactive system is being used, SET, RESET, CLS, TAB, PRINT@, POINT, POKE, and PEEK (TRS-80 examples) should be introduced. Proper X, Y coordinate display layout should be emphasized (two dimensional arrays).

184

185

6. identify and use the CASE statement;

The previous I/O features should be integrated with different problems using conditional statements and looping statements in problem solving.

Once students are proficient in programming simple tasks, using format to produce a readable and usable output should be stressed.

On page 30, "Problem Solving," it states that solutions to all problems can be expressed using the linear, branch, and loop constructs. It also states that the CASE statement can be used as a variation to the above. Pages 35, 45 and 46 illustrate examples of the CASE statement.

The IF-THEN-ELSE construct is used if there are two alternatives and one is to be chosen. If there are more than two choices then nested IF's can be used, but the CASE statement may be preferable as is illustrated in the following example.

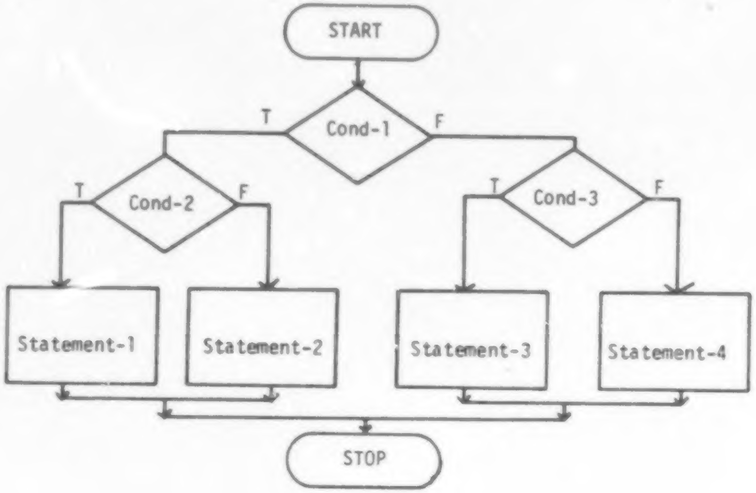
Example: With nested IF's and three conditional statements

```
IF condition-1
   THEN IF condition-2
         THEN statement-1
         ELSE statement-2
ELSE IF condition-3
         THEN statement-3
         ELSE statement-4
```

(There are different methods of indentation using the IF-THEN-ELSE.)

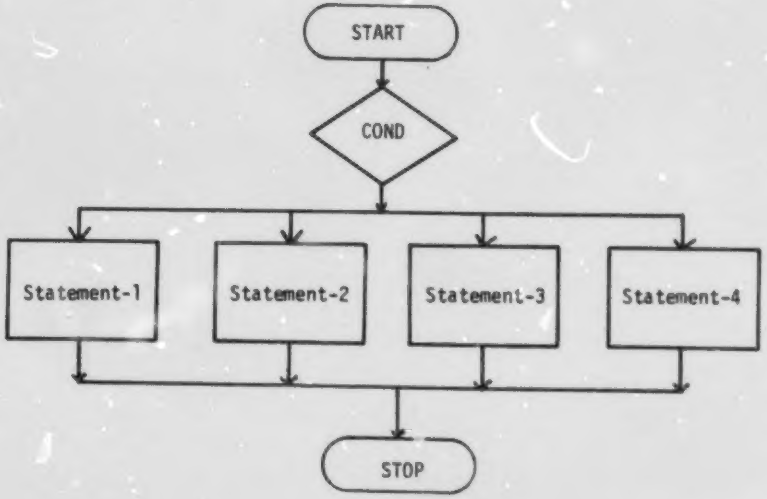
OBJECTIVES	NOTES AND EXPLANATION

The flowchart would be


START
Cond-1
T
F
Cond-2
T
F
Cond-3
T
F
Statement-1
Statement-2
Statement-3
Statement-4
STOP

(Statement-1, Statement-2, . . . could each be a block of statements rather than just one.)

With the CASE statement, compare the flowchart above to the example on page 35, "Problem Solving" or the one below.



(Statement-1, Statement-2, . . . could each be a block of statements rather than just one.)

Some people will find the above construct simpler to use than the nested IF construct. Nested IF's may also cause bugs in the program. Syntactically the nests may be constructed correctly (no error or warning messages) but do not do what they were intended. The CASE statement may, therefore, be preferable.

Basically, with the CASE situation, the computer can branch to any one of a number of instructions depending on the value of the variable or expression considered.

In BASIC the format is:

```
1)  ON (variable or expression) GOTO (label-1 . . . label-n)
2)  ON . . . GO SUB . . . .
```

```
260  IF B < 1 or B > 4 THEN GOTO 5000
270  ON B GOSUB 1000, 2000, 3000, 4000

1000 Rem Subroutine 1 when B=1
1010     S1=S1+A
1020     C1=C1+1
1030     RETURN
2000 Rem Subroutine2 when B=2
2010     S2=S2+A
2020     CS=CS+1
2030     RETURN
3000 Rem Subroutine3 when B=3
3010     S3=S3+A
3020     C3=C3+1
3030     RETURN
4000 Rem Subroutine4 when B=4
4010     S4=S4+A
4020     C4=C4+1
4030     RETURN
5000 Rem Error Code
5010     PRINT "Value of B is out of range"
```

Teachers are advised to stress indentation of subroutines and keeping blocks of code together (i.e., don't insert statements between 1030 and 2000 that are not dependent on the ON statement).

OBJECTIVES

NOTES AND EXPLANATION

In FORTRAN, the CASE statement can be:

1) a computed GOTO

Example: GOTO (99, 16, 47, 61), K

2) DO CASE control structure

Example: C CASE EXAMPLE

```
DO CASE VALUE
   CASE 1
      SUM1=SUM1+1
   CASE 2
      SUM2=SUM2+1
   CASE 3
      SUM3=SUM3+1
   CASE 4
      SUM4=SUM4+1
   IF NONE DO
      PRINT, "Value in Error", VALUE
END CASE
```

(Indentation should be stressed. Also, stress that blocks of code should be kept together.)

More teachers are probably familiar with the computed GOTO. The second structure may not be available on your system. The DO CASE does not need the GOTO's that are necessary between most statement segments of a computed GOTO.

In Pascal, it is called the CASE statement

OBJECTIVES

NOTES AND EXPLANATION

Example: (*COMPUTE NO. OF STUDENTS IN EACH LEVEL 1, 2, or 3*)

```
CASE LEVEL OF
      1:SUM1:=SUM1+1;
      2:SUM2:=SUM2+1;
      3:SUM3:=SUM3+1
END
```

In Pascal, the CASE does not test to see if LEVEL fits into the sequence. It has to be tested by the use of an explicit IF before the CASE statement.

With Pascal, the variable or expression evaluated does not have to follow a neat pattern of integer values, generally 1, 2, 3 and so on (or a multiple of this sequence). The variable or expression may fall into different categories or sequences depending on the way the variable is defined.

7. identify the need for simple data structures and to develop and to use simple data structures involving:
   a) an appreciation of the trend in computer programming from an emphasis on numerical computation to an emphasis on the storage and retrieval of data,

The computer was originally invented to speed up complicated and time-consuming numerical computations, such as ballistic tables required near the end of World War II. However, its use in the ensuing 25 years has led to its development as a tool to store and retrieve large quantities of information and, in fact, this new role has become dominant. Its ability to compute, that is, to calculate and to perform arithmetic, has in most applications become irrelevant.

These great quantities of information must be conveniently represented on the computer so that algorithms are easy to write, implement, and be debugged. Data structures in the language that is used provide this vehicle. Hence, the teacher should change emphasis from a stress on computations to a desire to represent data conveniently.

OBJECTIVES	NOTES AND EXPLANATION
b) consideration of the problem of what data best represent the information that is to be processed and then how to represent that data on the computer,	In solving a problem with or without a computer, it is first necessary to define the set of data that represents the real situation. If we think of the example of a student file at a school, every student is represented in the file by a set of data relevant to either the administration or to the teacher. This set includes some identification of the student (for example, name and marks), but likely will not include data such as height and hair color. The choice of what data should be included is influenced by the problem that is to be solved. Once this choice is made (usually this consideration is more important in file design), then the second challenge is how the data or information should be represented. This choice is influenced by the computer language being used. As well, this choice must be guided by the operations to be performed on the data. Some fundamental data structures are the array, the set, the record, and the file which are composed from base data structures such as integer, real, character, or logical (Boolean).
c) recognition that there are many ways of representing data, even simple non-structured data such as one student mark,	In this section, explanations concentrate mainly on the one-dimensional array (a vector) and the two-dimensional array (the table). The One-Dimensional Array Consider the problem of finding the arithmetic mean or average of a set of student marks. Student marks can be represented in three ways. For example, the mark 69, can be represented as: (i) the Character '6' followed by the character '9', (ii) the integer 69, or (iii) the real number 69.0 which permits fractional parts such as 69.7 which might be of assistance when deciding on a scholarship. (The student with 69.8 is slightly ahead of the student with 69.7. However, if integers were used, they would likely be represented as both having a mark of 70.)

OBJECTIVES

d) choosing a data structure dependent on the operations that are to be performed on that data making the problem much easier to solve,

NOTES AND EXPLANATION

When finding an "average," which means performing arithmetic on the values, the character representation in (i) is not suitable since it would have to be converted to a number before addition could occur. Using integers (as in (ii) is not suitable since one may be dealing with the awarding of scholarships and may require the extra precision not naturally given by integers. Hence, it is decided to use reals to solve the problem since the set of reals lends itself to computing an "average" quite readily. Note that the data structure chosen (REAL) has been affected by the operations that will be performed, that is: (i) finding an "average", and (ii) possibly ranking the students. A straightforward program in pseudo-code then becomes,

```
Sum is initialized to zero
Count is initialized to zero
WHILE more data DO
      Read in a VALUE
      Value is then added to Sum
      We increase Count by one
END WHILE
Compute Average as Sum divided by count.
```

Now suppose that the client for whom the program is being created wishes to know how much each mark deviates from the "average."

A first solution could involve the following. Suppose there are 10 marks. Call them A, B, C, D, . . ., J and then after these values are read in, compute the sum.

```
Sum is A+B+C+D+E+F+G+H+I+J
```

This is clumsy and inflexible if there is a need to vary the number of students for whom we are computing the average and the separation of individual marks from the average. A new operation and the old data structure, that is, individual variables representing each mark, is not suitable. Hence, look for a new data structure or invent one, if necessary. Fortunately, all high level

OBJECTIVES	NOTES AND EXPLANATION

e) description of how an array is stored internally, and how an individual element is accessed,

programming languages provide the one-dimensional array data structure, where if we have N marks, they are stored in M(1), M(2), . . . , (M(N).

The array is possibly the most widely known structurer of data because in many languages including FORTRAN and ALGOL 60, it is the only explicitly available structure.

An array is a homogeneous structure; that is, its components are all of the same type. In the example of marks, they are all real numbers. The first one can't be integer, the next character, the next real and so on (this facility is partially provided by the RECORD structure in Pascal, for example). Another feature of the array is that it is a random-access structure, that is, all elements can be selected at random and each element is equally accessible (for example, it does not matter if we wish to get M(4) or M(7). In a sequential file to get M(7) first read over M(1) to M(6)). Visualize the array in the following manner:

M(1)	79.6
M(2)	83.7
M(3)	49.2
M(4)	99.6
M(5)	87.2
M(6)	40.1
M(7)	67.3

Notice that each element of the array is in memory directly after the previous one (contiguous memory occupation). If M(1) is assigned an address in memory (the start of the array) say A, then the address of the jth component in memory is A + j x s where "s" is the amount of storage occupied by one element of the array M. The compiler automatically provides statements that accomplish this, and frees the programmer to accomplish more work and to think about the problem at a higher level, without this annoying attention to detail.

BEST COPY AVAILABLE

OBJECTIVES	NOTES AND EXPLANATION
	Another important point is that array indices may be computed from an expression. For example, if we wanted M(6) we could say M(2 + 4) or M(2 + 2 x 2). Notice that this expression is substituted in place of an index constant. The expression then evaluated and the result determines the selected component. This generality gives the programmer a powerful and significant programming tool. However, it also gives rise to one of the most frequently encountered programming mistakes: the resulting index value may be outside the interval specified as the range of indices to the array. An adequate compiler should provide a warning if you try to access a nonexistent component. In the example, if M(i*j) is asked for where i has value 10 and j has value 2, then you are attempting to access component M(20). If only 7 components have been set aside for array M, then asking for M(20) makes no sense.
f) recognizing how a one-dimensional array makes the solution to some problems possible and makes other problems easier to solve,	Now, return to solving the problem of determining the deviation of each mark from the "average" mark. The program is simple and can be roughly expressed as:

```
Input the number of marks into N
Set I to 1
Set Sum to 0
WHILE I is less than N DO
      Read a value into M(i)
      Add M(i) to Sum
      Increase I by 1
END WHILE
Average is the Sum divided by N
Set I to 1
WHILE I is less than N DO
      Print the absolute value of M(i) - Average
      Increase I by 1
END WHILE
```

OBJECTIVES	NOTES AND EXPLANATION
	Notice that the use of the data structure, the array, has greatly facilitated the solution to the problem. The same is true for the record, the set, the file, pointer variables or, as well, other user-defined data structures. It is suggested that if the correct data structure is found, the corresponding algorithms and the operations that the algorithm must perform are greatly simplified (or even made possible). The job of the programmer or systems analyst is to be aware of suitable data structures (tools of the trade), to be able to create new data structures, and to know how to apply them appropriately.
g) appreciation that a particular data structure permits certain operations to be carried out efficiently,	The teacher should have a summary of possible operations that can be performed on a one-dimensional array. The following is a partial list which can be lengthened. Assume the array M(1), M(2), . . . , M(N) throughout. (i) Find the sum of the elements and the average. (ii) Find the largest or smallest element. (iii) Sort the elements in ascending or in descending order. (iv) Look up an element in an array. (v) Add/delete an element to/from an array. If you wish to insert it between M(1) and M(N), say between M(3) and M(4), all the elements from M(4) on down must be moved one place to the right. A similar operation applies if you wish to delete an element (e.g., get rid of the value in M(2)). The value in M(3) is shifted to M(2), M(4) to M(3) and so on. (NOTE: For large arrays this is an inefficient operation and means that the data structure should not be used if this operation is required frequently.) (vi) Rotate the elements in the array in a positive or a negative direction.

OBJECTIVES

NOTES AND EXPLANATION

Example: a positive rotate by 1 is
M(1) goes to M(N)
M(2) goes to M(1)
M(3) goes to M(2)
and finally
M(N) to M(N-1)

(vii) Interchange two elements of an array, say M(i) and M(j).

(viii) Perform partial sums, say

S(1) = M(1)
S(2) = M(2) + M(1)
S(3) = M(3) + M(2) + M(1)
.
.
S(N) = M(N) + M(N - 1) + . . . + M(1)

(ix) Initialize the elements of M to zero or some constant or some sequence. For example, M(1) = 1, M(2) = 2, . . . , M(N) = N

(x) Flip the elements of the array

M(1) changes place with M(N)
M(2) changes place with M(N-1)
M(3) changes place with M(N-2)
. . . and so on to the middle of the array.

This is only a partial listing of the possible operations on a one-dimensional array. If a new operation is required then the questions to be asked are: Is the one-dimensional array convenient? If the one-dimensional array is suitable, can the new operation be created out of a combination of the listed operations?

OBJECTIVES	NOTES AND EXPLANATION
h) recognition of the need for a two-dimensional array,	The Two-Dimensional Array

Again, motivate the requirement for a two-dimensional array through a problem that requires certain new operations to be performed on the data. Suppose that a client indicates that each student is taking five subjects and that the "average" for each student (an operation on the data structure) is to be computed. The program is a modest variation of the one required to compute the class average. In pseudo-code, write:

```
Input the number of students N
Set I to 1
WHILE I ≤ N DO
      Set J to 1
      Set Sum to 0
      WHILE J ≤ 5 DO
            Read a value into M(i)
            The value for M(i) is added to Sum
            I is incremented by 1
      END WHILE
      Print out that the average for student I is SUM divided by 5
      Increment I by 1
END WHILE
```

The client, after getting the results from this program (after it works on the computer) is delighted and now sees many possible new applications and operations on the data. He now decides that he would like the overall class average, and the average for each course. This can be readily accomplished by creating a one-dimensional array of length five to hold the sums for each course before their average is computed. At this point, new operations on the data have been accommodated fairly successfully. However, note that two-dimensional array (a new data structure) could accomplish these new operations in a more straightforward manner; that is, more clearly to the programmer and to persons reading the program. A new data structure can be forced if our client indicates that for each student, he would like to know the deviation of each mark from the

OBJECTIVES	NOTES AND EXPLANATION

average mark in the course as well as the deviation of each mark from the overall average. Basically, our new operations require that each mark be in a table and be instantly accessible. Really, the operation required is to be able to look at any student's mark at any point in time. Thus, the two-dimensional array is nothing more than a table representation with two subscripts; one to point to a row, the other, to point to a column.

Hence, when referring to the i'th student's mark in course j, simple say: MARK (i, j).

i) description of how a two-dimensional array is stored internally;

It is important to know how data structure is stored or represented in the computer. Let us assume that MARK is a 2 by 3 table (two students each with three marks). As well, let us assume that the table is stored linearly in memory column by column. Some languages store tables row by row and the calculation required to access an element in the table (its address) will be slightly different. Memory will then look as follows (column by column storage is assumed):

MARK (1,1)	67.2
MARK (2,1)	96.3
MARK (1,2)	87.2
MARK (2,2)	66.3
MARK (1,3)	98.1
MARK (2,3)	72.6

The first address in the table (its location in memory) is known and is assigned by the compiler; let us say that it is A. How does the computer determine where Mark(1,3) is? We can note that it is four words past MARK(1,1). After some effort, we find that MARK(i,j) can be found using the formula:

A + ((j-1)*(no. rows in table) +i-1) * word length
If we put in i=1 and j=3, we find
A + ((3-1) x 2 + (1-1) * word length = A + 4*word length

OBJECTIVES	NOTES AND EXPLANATION

that is MARK(1,3) is four words past the start of the table, which is exactly the result obtained by "eyeballing" the storage representation. At this point, we note that the compiler handles this detail for the programmer, which is an advantage to using a high-level language over a low-level languge such as Spectre or Spectre Map where the programmer has to be concerned with the detail.

A first approximation to an algorithm to solve our client's problem follows: the teacher should stress getting a working program before any optimization (memory time saving attempts) occurs. This is the approach used in the following program. As well, assume that computerization involves only the overall average and the difference of each course from this average.

```
Input the number of students N
Input the number of courses M
Set Sum to 0
Set index I to 1
WHILE I ≤ N DO
   Set index J to 1
   WHILE J ≤ M DO
       Input a value for Mark (i,j)
       Add Mark(i,j) to Sum
       Add 1 to J
   END WHILE
   Add 1 to I
END WHILE
The overall Average is Sum divided by M x N
```

Calculate and print the difference of each mark from the overall average.

The last part of the solution has been described as a statement that can be thought of as a high-level command. This indicates another important point concerning program design. We have commented on the detailed calculations required to access an element MARK(i,j) in an assembly or a machine language.

OBJECTIVES	NOTES AND EXPLANATION

The data structure, the two-dimensional array in a higher-level language, has freed the programmer from all the detail. A parallel situation occurs in program design. The preceding solution can be summarized as follows:

(i) input the table of marks;
(ii) calcualte the average of the table of marks;
(iii) calculate the deviation of each mark from the average;
(iv) print out the average mark and the deviation of each mark from the average.

This expresses the steps in the solution in a high-level manner. The details of the solution can now be filled in progressively (called 'step-wise refinement'). This is an important technique in programming and should be treated in a section on program design and implementation if the teacher either has the time or can fit this concept in as sample programs are worked in class.

Some of the operations that can be performed on a two-dimensional array are listed below.

(i) Sum the rows and columns of a table and compute corresponding averages.
(ii) Find the largest and smallest elements in rows, columns, or the complete table.
(iii) Sort individual rows or columns of a table.
(iv) Do a table lookup.
(v) Interchange two rows of two columns of a table.
(vi) Initialize table entries to particular values (all zero, to the identity matrix, etc.)
(vii) Perform some linear algebra type operations:
  . add/subtract two tables of values;
  . multiply rows or columns of a table by a constant (or whole table);
  . add a multiple of one row (column) to another row (values);
  . multiply one matrix times another;
  . solve a system of equations (library routine).

OBJECTIVES

8. create, manipulate and retrieve files.

NOTES AND EXPLANATION

Files as Data Structures

Arrays are data structures which are by definition homogeneous throughout. A file is a more general data structure in which elements of arbitrary, and themselves possibly structured, types are joined together into a group of data. One group of such data is called a record. The individual elements in the record are called fields, and a group of records are a file.

Storage Techniques

Files can be categorized into one of two types based on how they are stored. All tape files and many disk files are stored sequentially. The sequence is based on one or a combination of fields in a record. For example, a file of student records could be stored in ascending order by student number. Another alternative would be to store them in ascending order by student names. The term "sequence" here implies that the records themselves are generally stored in a physical sequence on the tape or disk.

Files stored on direct access storage devices (usually disk) can also be stored in random fashion. As records are added to a file, no regard is given to any "index" field in a record to attempt to keep records in any order. The next record is simply placed in any available location on the disk.

Access Techniques

Sequential files can only be accessed in a sequential manner. If the required record is record number 1000 of a one thousand record file, then every record of the file must be read to obtain the required information. If a sequential file is to be updated, every record must be read until the required one is located, it is updated, and then all the rest are read. In each case, after reading a record, it must be written to a new file which is still in strict sequential order.

OBJECTIVES	NOTES AND EXPLANATION

Sometimes files which are normally accessed sequentially also need to be accessed on a random basis. The access method is referred to as indexed sequential. The file is in some type of sequential order when it is first created, but the computer then creates a special index on the disk which makes it possible to access a specific record without reading all the records leading up to that one. As new records are added to the file, space is found for them on the disk where available, and the place recorded in the index.

Pure random, or dynamic, files do not attempt to store data in sequential order. An index is kept of all records, but in this case the index is created as records are written to the disk. The indexed sequential file is a little more efficient and is preferred if there will be few additions and deletions from a file.

Program Files on a MICRO

After a program has been entered into a micro's memory (e.g., a PET), the user may wish to save it.

To save a program on a cassette, proceed as follows:

(i) Place a cassette in the recorder and ensure that the recorder is attached to the computer.

(ii) Type SAVE on the keyboard.
- The program will be stored on the cassette without a name.
- It is better to use SAVE "progname" where progname is the name the programmer has given the program. Then the programmer will be able to ask for the program specifically at a later date.
- SAVE "progname", 1, 2 has another advantage in that it writes an end-of-tape marker after the program itself. If more programs are to be saved on the tape, then it is possible to have the tape stop at the EOT mark and then start saving the next program.

OBJECTIVES	NOTES AND EXPLANATION

(iii) A PET responds to the SAVE command with

```
PRESS PLAY & RECORD ON TAPE #1
```

After the user has done this, the PET will respond with

```
OK
WRITING
```

After the save is complete, the PET will respond with

```
READY
```

(iv) To ensure that the SAVE has been done properly, the user can request that a VERIFY be done.

. Type VERIFY "progname".
. The PET responds with:

```
PRESS PLAY ON TAPE #1
OK
SEARCHING
FOUND "progname"
VERIFYING
```

. If the comparison is not exact then a ?VERIFY ERROR message is given. If this occurs it is necessary to go through the SAVE procedure again.

Loading a program from cassette:

After a file has been loaded and properly verified, it may be retrieved from the tape and placed back in the computer's memory.

OBJECTIVES	NOTES AND EXPLANATION

(i) LOAD will cause the first program on the rewound tape to be loaded.
(ii) LOAD "progname" will cause a search for the program "progname." The computer indicates that it is SEARCHING, and keeps the user informed of the files that it has found.
(iii) If a ?LOAD ERROR message is displayed, then it is necessary to rewind the tape and try the LOAD again until successful.

Program Files on Diskettes

In general the logic of saving and loading files on diskette is the same as for cassette. However, the syntax involved becomes much more system dependent. For example, on the PET it depends on the version of BASIC (i.e., 1.0, 2.0, 3.0, or 4.0) and the version of the disk operating system (i.e., 1.x or 2.x). Therefore, more detailed information on this topic must be obtained from the appropriate manuals and guides.

Using Files in Programs

Files are stored on disk (or tape) by name. Depending on the system, this name may be six or seven alphanumeric characters (restrictions vary) and in some cases also include a three-character extension, separated from the name by a period (e.g., JOHN.TXT).

In writing a program, these files are sometimes assigned a number (called a channel designator), and thereafter referred to by this number. This is true in BASIC and FORTRAN. Other languages, like COBOL, assign a local name to the file, and then reference it using this local name.

Therefore, to use a file, it must first be "opened." In BASIC, this may look like:

```
OPEN "JOHN.TXT" AS FILE 1
```

OBJECTIVES	NOTES AND EXPLANATION

Then to obtain informtion from this file, you would use:

```
INPUT #1,A,B,C
```

Similarly, to output information to the file, you would use:

```
PRINT #1,A,B,C
```

At the end of the run, before the program terminates, all files which were opened should be closed. Failure to do so can cause loss of information on those files. A typical close statement may be:

```
CLOSE#1
```

These descriptions are general in nature. Details will be different for some systems and will also depend on the language being used. The teacher must refer to the manuals for the equipment being used for specific information.

IV. TECHNIQUES

A. Debugging

OBJECTIVES	NOTES AND EXPLANATION
The student should be able to:	
1. detect and correct programming errors involving syntax;	Looking for (and fixing) program bugs or mistakes is called debugging. Most debugging should take place while a program is still on paper, before it is entered, compiled, and run. A new term that has surfaced is "antibugging," which means programming in a way that helps avoid making bugs in the first place. Since many programming shops spend as much as 50 percent or more of their time on debugging, the two terms assume considerable importance.
2. detect and correct programming errors in logic.	Bugs fall into two categories: syntax bugs, meaning that the rules of the language have not been followed; and logic bugs, indicating that the logic is at fault. Every attempt should be made on the part of the instructor to obtain a good "debug compiler" or "debug interpreter" that assists the beginning programmer to find mistakes in a program. Common Programming Errors A list of common programming errors should help the student with the debugging process and make him/her less dependent upon the teacher. Following is such a list. a) Use of undefined variables: . variable has not been initialized; . name has been changed inadvertently during the course of the program; . a zero is punched instead of an alphabetic O.

OBJECTIVES	NOTES AND EXPLANATION

b) Use of wrong data type (REAL instead of INTEGER).
c) Provision of insufficient data. (The error message for this is usually something like "INVALID DATA" which is confusing. This is caused because an end-of-file mark has been encountered.)
d) Omission of a branch instruction.
e) Looping back to the incorrect place. This can cause an infinite loop due to re-initialization of the variables.
f) Subscripting out of bounds. This occurs because the looping controls have gone beyond the limit of the DIMENSION statement.
g) Dividing by zero or taking the square root of a negative.
h) Using an incorrect argument type in a pre-defined function.
i) Stopping a looping procedure incorrectly by going through the loop one too many or one too few times.
j) Incorrect order of operations in a mathematical expression.
k) Unexpected truncation, caused by integer division or assignment to an integer memory location.
l) Improper print formats (mixing up variable type and format type).
m) Mismatched parentheses.
n) Incorrect results from complicated Booleam expressions due to confusion over order of operations involving NOT, AND, OR.
o) Improper nesting.
p) Misspelling keywords.

Teachers generally attempt to provide the correct model by presenting programs without bugs. However, programs that have bugs should also be given to students so that they can develop ability in debugging.

It is vital in starting the debugging to define the problem initially as completely as possible. Construct sample output that the program should give. This data will form the basis of testing--that is, validating the program on given sets of data. It is important to remember that testing can show the presence of bugs but never their absence. The program test is only as realistic as the test data. Construction of sample input and output will ensure that the problem is understood initially before programming begins. Hence, before the program is written, the programmer has created sample input and resulting output and has done some sample calculations by hand. This will assist the programmer in following the program logic when reading over the listing as the program is being corrected. The programmers should make sure that the test data includes boundary conditions. For example, if it is a sort program, it should sort 0, 1, or n elements.

Influence of Debugging on Coding

Debugging starts as you code the program solution. Print statements echo the input. Many people try to correct a program that is working on erroneous data. Additionally, comments that delineate what a block of code is supposed to do are helpful. Comments that delineate what each line is doing are not useful and are not changed by the programmer when the program is changed, thus giving a false impression of what the code is supposed to do rather than what it does. Blank lines left between logical units are also very helpful.

Elimination of Syntax Errors

Let us assume that you have a program that is executing. The syntax errors (the errors where you have not followed the rules of the language) are easiest to remove with the assistance of a good compiler.

Desk Checking

Checking the program by hand, with the data that you used to clarify the problem definition, can now be used to check each logical block in the program. Studies have shown that programmers working alone can debug a program just as fast on average as using the computer to assist the debugging effort, and therefore the so-called "desk check" is a productive start. If you think that you have found the error, proceed with caution when making the change, as every correction has a 50 percent chance of working.

OBJECTIVES	NOTES AND EXPLANATION

Review Program with Another Person

Another excellent debugging technique is to explain the program to someone else. As you explain the program to another person, an interesting phenomenon takes place: you seem to look at your own solution in a refreshingly new manner and often find your own mistake!

Sleep On It

Another solution (if you have started the debugging effort soon enough) is to leave the program overnight. The programmer's mind will work on the solution during that time and when he/she next looks at the program fresh thoughts may come to mind.

Use Debug Prints

Another useful technique is to trace the program by having values of critical variables printed as each statement executes.

Vary the Input

With larger programs and difficult bugs, a useful technique which often sheds some light on why the program is malfunctioning is to vary the input and observe the output.

Debugging Larger Segments

In larger programs, it is useful to print various partial results to isolate the area where the problem is occurring. Try to determine if the bug lies in the first half or the seacond half of this area. Once determined, divide this new area in half and repeat this effort until the problem is isolated to a distinct statement or a block of statements. This is called a binary debug technique.

Rewriting Rather Than Patching

Often, as the debugging proceeds, one finds that in the initial definition on the solution, there were several points that were missed. Sometimes it is better at this stage to use this new-found knowledge to rewrite the program. Putting one patch on top of another in a poorly designed program can often consume much more time than a rewrite. A rewrite can be done amazingly quickly with new-found knowledge.

Teachers could give students a program which needs to be patched. Hopefully, after attempting to patch the program, students will see the advantage of rewriting the program.

Error Diagnostics

The teacher could get copies of the error diagnostics most commonly found in programs and go over them with students. It is preferable that students be strongly encouraged to read error diagnostics rather than leaving them for the teacher to decipher.

B. Characteristics of a Good Program

OBJECTIVES

The student should be able to:

1. identify good programming techniques;
2. use good programming techniques.

NOTES AND EXPLANATION

It is suggested that teachers provide samples of good programs for the students' perusal. The following is a checklist to follow.

Characteristics of Good Programming Techniques

1. The test data must be thorough enough to support the opinion that the program works.
2. The program should be written so that it minimizes the debugging and testing time. Consider the following:
   a) Debug statements may be used as part of the original program - that is printing out partial results of computations as a matter of course.
   b) Variables should be checked to ensure that they are in the desired range. This could prevent such errors as subscript range violations and division by zero.
   c) The code should be well documented. At the head of the program, several comments should indicate who wrote the program, when it was written, and for whom. Then include a short comment indicating what task the program performs. The variables used in the program should be listed in alphabetical order with a comment indicating what each one does. Also, within the program, a comment should begin each block of code.
   d) Good mnemonic names should be used in the program to assist readability. One suggestion is to leave out the vowels in a variable name. Another suggestion is that long names be used for important variables

OBJECTIVES	NOTES AND EXPLANATION

and shorter names for less important variables such as loop indices. The teacher should hand out samples of programs that are representative.

e) The program should avoid the use of GOTO's if possible. If they are used, they should be used in a structured fashion and every attempt be made to avoid spaghetti-bowl logic.

f) A simple program style should be used. Check the logical expressions for nesting more than two-deep, or compound logical expressions that can be simplified. Ensure that the program can be understood by others.

g) The program body should use appropriate indentations, spacing, and blank lines. This practice assists the programmer by readily indicating the code that is dependent on a particular control structure such as an IF statement. It also increases program clarity, readability, and maintainability.

h) Programs should be constructed using the three building blocks of structured programming, namely:

  i) linear or simple sequence,
  ii) selection process (IF-THEN-ELSE type construct),
  iii) repetition or loop (PERFORM type construct).

i) Program modules should have only one entry and one exit point.

j) Program modules should be kept to a manageable size of approximately 50 to 60 lines.

3. The program should be flexible. This implies that it should be easy to change, expand, or upgrade. For example, ask:

   a) What happens if the table size is to be expanded?
   b) What if more transaction codes are to be processed?

   Since most programs are to be changed in their lifetime, design with this in mind.

4. Examine the program to determine if an appropriate algorithm has been selected. For example, in searching an ordered list of items a binary search would be more appropriate than a liner search.

5. The program is efficient in memory and execution time. This is falsely considered by many programmers to be the most important part of the program. Many programmers spend much of their time writing optimized code that tends to be obscure. It is far better to get the program running and then run a profile to find out which statements are executed most often. It has been found that most programs spend 90 percent of their time in 10 percent of the code. Hence, the optimization effort can be devoted to this 10 percent area.

appendices

APPENDIX A

SPECTRE OPCODES

Definitions of instructions given by the two-digit operation code:

+0000010WWW	WWW is copied into AC, previous AC lost, WWW is unaltered
+0000011WWW	AC copied into WWW, previous WWW lost, AC is unaltered
+0000012WWW	WWW coied into MQ, previous MQ lost, WWW is unaltered
+0000013WWW	MQ copied into WWW, previous WWW lost, MQ is unaltered
+0000015WWW	causes +0000000000 to be stored in WWW
+0000017WWW	changes sign of AC to the opposite sign ('+' to '-', etc.)
+0000020WWW	WWW is added to AC, previous AC lost, WWW unaltered
+0000021WWW	WWW is subtracted from AC, previous AC lost, WWW unaltered
+0000022WWW	MQ is multiplied by WWW, result placed in AC and MQ, with least sig. figs. in MQ, previous AC and MQ lost, WWW unaltered
+0000023WWW	20 digit in AC and MQ is divided by WWW, with result stored in MQ, remainder stored in AC, previous AC and MQ lost, WWW remains unaltered
+0000030WWW	data card is read and placed in WWW
+0000031WWW	WWW is printed after spacing to a new line, WWW is unaltered
+0000040000	computer ceases execution
+0000050WWW	transfers control back to instruction WWW
+0000051WWW	transfers control to WWW if AC is negative or zero
+0000052WWW	transfers control to WWW if AC is non-zero
+0000053WWW	transfers control to WWW if AC is positive (or zero)
+0000054WWW	transfers control to WWW if AC is zero

SPECTRE OPCODES

NUMERIC OPCODE	MNEMONIC	NUMERIC OPCODE	MNEMONIC
10	CLA	23	DIV
11	STO	30	INP
12	LDQ	31	OUT
13	STQ	40	STP
15	STZ	50	TRA
17	CHS	51	TLE
20	ADD	52	TNZ
21	SUB	53	TPL
22	MPY	54	TZE

APPENDIX B

STRUCTURED PROGRAMMING

1. Definition

Structured programming is a programming style tht uses a well-disciplined approach to writing programs. This style reduces a program's complexity, resulting in increased clarity and easy maintenance. Structured programming uses three basic constructs:

a) linear sequence
b) selection, e.g., IF...THEN...ELSE
c) repetition or loop
    e.g., PERFORM...UNTIL
        REPEAT...UNTIL
        FOR...NEXT

In addition, there are related techniques to the above three constructs that when used judiciously contribute to structured programming. These are:

a) routines or segments of code having only one entry and one exit point;
b) programs which proceed in a top-down fashion;
c) indention, blank lines, and comments, which contributes to program clarity, readability and maintenance;
d) avoiding putting multiple statements on one line;
e) including a dictionary of variable names.

2. History

In a letter he wrote to ACM in 1973, entitled "GOTO Statement Considered Harmful," Edgers Dijkstra started the structured programming revolution. Bohm and Jacopini followed by setting the theoretical framework. They stated that any program could be written using only three basic constructs or structures:

a) linear sequence;
b) selection;
c) repetition or loop;

3. Why Structured Programming?

A student's computer program is the written record of his or her solution to a problem. A good analogy can be made with regard to writing an essay or short story. Before students write an essay they are taught to spell, to write a sentence, build a paragraph, and to organize the basics into the logical form of an essay. In other words, they are taught the basics before the task of producing an essay is attempted.

On the other hand, although students are taught the syntax of a computer language they are not shown how logically to organize a program. As a result they end up putting statements together in an unorganized form, thus producing "SPAGHETTI" code.

It is no wonder that an unstructured program of more than 30 lines is difficult to understand, debug, and change. As well, teachers will have a hard time reading and critiquing the program, and will end up grading the result only by whether it works or not.

On the other hand, although students are taught the syntax of a computer language they are not shown how logically to organize a program. As a result, they end up putting statements together in an unorganized form, thus producing "SPAGHETTI" code.

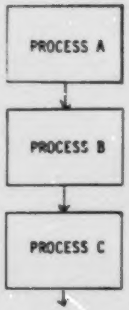
It is no wonder that an unstructured program of more than 30 lines is difficult to understand, debug, and change. As well, teachers will have a hard time reading and critiquing the program, and will end up grading the result only by whether it works or not.

4. Structured Programming and BASIC

In BASIC or for that matter any high-level language, it is possible to use only the three basic constructs. However, in most versions of BASIC the Selection and Repetition processes must be simulated since you do not have an IF...THEN...ELSE statement or a WHILE... DO statement. In addition, to use indentation as a means of increasing clarity is not convenient on the Apple and PET microcomputer (i.e., you must use colons to force indentation).

a) Linear Sequence

This construct is possible in all high-level languages.

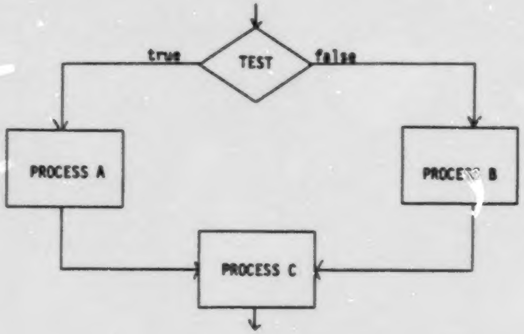


Example

```
10 REM   THIS PROGRAM ADDS 2 NUMBERS
20 REM
30 REM   DICTIONARY
40 REM       A = INPUT VALUE
50 REM       B = INPUT VALUE
60 REM       C = A + B
70 REM
80 PRINT "HELLO"
90 PRINT "ENTER 2 NUMBERS AND I WILL ADD THEM"
100 INPUT A,B
110 C = A + B
120 PRINT A;"+";B;"=";C
130 END
```

b) Selection

Since there is not IF...THEN...ELSE only an IF... THEN in most versions of BASIC, we must simulate this process.

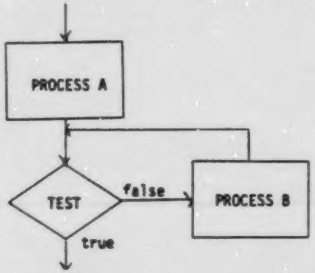


Example

```
10 REM     PAYROLL PROBLEM
20 REM
30 REM     DICTIONARY:
35 REM       H = HOURS
40 REM       R = RATE OF PAY
45 REM       P = GROSS PAY
50 REM
60 INPUT "ENTER HOURS AND RATE"
70 INPUT H,R
80 IF H > 40 THEN 100
90 GOTO 120
100     P = 40 * R + (H-40) * 1.5 * R
110     GOTO 140
120 REM ELSE
130     P = H * R
140 REM ENDIF
150 PRINT H,R,P
160 END
```

c) Repetition or Loop

Since there is not WHILE...DO or REPEAT...UNTIL in most versions of BASIC, we must simulate this process.



Example

```
1  REM     SQUARE ROOT PROBLEM
5  REM
10 REM     DICTIONARY
15 REM       N = VALUE INPUT
20 REM       I = FOR LOOP VALUE
25 REM       SQ = SQUARE ROOT
27 REM
30 PRINT "ENTER A NUMBER"
40 INPUT N
50 REM
60 IF N < 0 THEN 150
70       PRINT "NUMBER", "SQUARE ROOT"
75       PRINT "------", "-----------"
80       FOR I = 1 TO N
90           S = SQR(I)
100          PRINT I,S
110      NEXT I
120      PRINT "ENTER A NUMBER"
130      INPUT N
140      GOTO 60
150 REM END LOOP
160 END
```

SAMPLE PROBLEM

1. a) Given the hours and pay rate, write a program to calculate the gross pay. Use zero to terminate the program.

Unstructured Solution

```
10 PRINT "ENTER HOURS AND RATE"
20 INPUT H,R
30 IF H = 0 THEN 70
40 P = H * R
50 PRINT H,R,P
60 GOTO 10
70 END
READY
 RUN
ENTER HOURS AND RATE
? 33,10
 33             10              330
ENTER HOURS AND RATE
? 45,8
 45             8               360
ENTER HOURS AND RATE
? 0,0
READY
```

255

Structured Solution

```
10 PRINT "ENTER HOURS AND RATE"
20 INPUT H,R
30 IF H = 0 THEN 90
40      P = H * R
50      PRINT H,R,P
60      PRINT "ENTER HOURS AND RATE"
70      INPUT H,R
80      GOTO 30
90 REM END LOOP
100 END
```

b) Modify the program to pay time-and-a-half for any hours greater than 40.

Unstructured Solution

```
10 PRINT "ENTER HOURS AND RATE"
20 INPUT H,R
30 IF H = 0 THEN 90
35 IF H > 40 THEN 70
40 P = H * R
50 PRINT H,R,P
60 GOTO 10
70 P=40*R+(H-40)*(R*1.5)
80 GOTO 50
90 END
```

256

```
READY
>RUN
ENTER HOURS AND RATE
? 30,11
 30               11                330
ENTER HOURS AND RATE
? 40,10
 40               10                400
ENTER HOURS AND RATE
? 41,10
 41               10                415
ENTER HOURS AND RATE
? 45,10
 45               10                475
ENTER HOURS AND RATE
? 0,0
READY
```

NOTE: The arrows in solutions (b), (c), and (d) show the poor control structures of the programs. The control branches up and down. There is no attempt to organize the program in a logical form. There is not indentation or single entry-single exit control structures. Solution (d), although relatively short (14 lines), is difficult to follow. Imagine the readability problem if this were a program of 30 lines.

Structured Solution

```
10 PRINT "ENTER HOURS AND RATE"
20 INPUT H,R
30 IF H = 0 THEN 90
32     IF H > 40 THEN 36
34     GOTO 40
36        P = 40 * R + ( H-40) * R * 1.5
38        GOTO 44
40     REM ELSE
42        P = H * R
44     REM ENDIF
50     PRINT H,R,P,
60     PRINT "ENTER HOURS AND RATE"
70     INPUT H,R
80     GOTO 30
90 REM END LOOP
100 END
```

NOTE: The solutions (a), (b), (c), and (d) present logical and readable solutions to the problems. Note the control structure (lines 32 to 44) illustrating an IF...THEN...ELSE. By setting the structure up in this form, there is a single entry and single exit point. This is not the case in the unstructured solutions.

c) Modify the program to calculate income tax at 25%.

Unstructured Solution

```
10 PRINT "ENTER HOURS AND RATE"
20 INPUT H,R
30 IF H=0 THEN 90
35 IF H>40 THEN 70
40 P = H * R
45 T=P*.25
50 PRINT H,R,P,T
60 GOTO 10
70 P=40*R+(H-40)*(R*1.5)
80 GOTO 45
90 END
READY
>RUN
ENTER HOURS AND RATE
? 40,10
 40              10              400             100
ENTER HOURS AND RATE
? 50,8
 50              8               440             110
ENTER HOURS AND RATE
? 0,0
READY
```

Structured Solution

```
10 PRINT "ENTER HOURS AND RATE"
20 INPUT H,R
30 IF H = 0 THEN 90
32     IF H > 40 THEN 36
34     GOTO 40
36        P = 40 * R + ( H-40) * R * 1.5
38        GOTO 44
40     REM ELSE
42        P = H * R
44     REM ENDIF
46     T = P * .25
50     PRINT H,R,P
60     PRINT "ENTER HOURS AND RATE"
70     INPUT H,R
80     GOTO 30
90 REM END LOOP
100 END
```

d) Modify the program to calculate income tax at 25% for the first $300. and at 30% for any amount greater than $300.

Unstructured Solution

```
10 PRINT "ENTER HOURS AND RATE"
20 INPUT H,R
30 IF H=0 THEN 90
35 IF H>40 THEN 70
40 P = H * R
45 IF P > 300 THEN 85
47 T=P*.25
50 PRINT H,R,P,T
60 GOTO 10
70 P=40*R+(H-40)*(R*1.5)
80 GOTO 45
85 T=300*.25+(P-300)*.30
87 GOTO 50
90 END
READY
>RUN
ENTER HOURS AND RATE
? 30,10
 30             10              300             75
ENTER HOURS AND RATE
? 40,10
 40             10              400             105
ENTER HOURS AND RATE
? 0,0
READY
```

Structured Solution

```
10 PRINT "ENTER HOURS AND RATE"
20 INPUT H,R
30 IF H = 0 THEN 90
32     IF H > 40 THEN 36
34     GOTO 40
36        P = 40 * R + ( H-40) * R * 1.5
38        GOTO 44
40     REM ELSE
42        P = H * R
44     REM ENDIF
46     T = P * .25
48     GOTO 54
50        T = 30 * .25 + (P-300) * .30
52        GOTO 58
54     REM ELSE
56        T = P * .25
58     REM ENDIF
59     PRINT H,R,P,T
60     PRINT "ENTER HOURS AND RATE"
70     INPUT H,R
80     GOTO 30
90 REM END LOOP
100 END
```

APPENDIX C

CONDITIONAL STATEMENTS

Ways of expressing a conditional statement include the use of a GO TO and the use of an IF-THEN-ELSE. We illustrate the first case (GOTO) using a pseudo code example that might arise in cooking (many programming constructs can be illustrated using real-life examples to illustrate a point rather than adding the complexity of programming syntax to the example).

```
     if (hard) go to 11
     if (tall) go to 12
     if (juicy) go to 13
     roast stop
11   if (green) go to 14
     peel grill stop
12   chop fry stop
13   boil stop
14   peel roast stop
```

The main problems with this use of the GOTO to implement the conditional statement include:

1. If this program stops why did it stop? Since there are many locations that the stop took place, it is difficult to trace through the logic to determine the conditions causing the stop.

2. Also, it is not immediately clear when you get to a particular statement, what conditions are in effect. For example, if you are at 12, did something in 11 cause an eventual branch to 12 (the logic could have gone to 14 and then, later on, branched back to 12)? It appears that only if the first line is executed, do you get to 11. However, in a larger program you can never be sure. This ambiguity has motivated the structured programming revolution. It has often been stated that the number of bugs in a program is proportional to the number of GOTO's. (The converse, that if there are no GOTO's then there are no bugs, is not true.)

Through the use of nested IF's, many objections can be overcome. (Also, we have the basis for the structured programming revolution, that is, the elimination of unnecessary GOTO's.) Hence we have (similar construction to Pascal and structured WATFIV):

```
if (hard) then
   Begin peel
     If (green) then
        Begin roast
        end
     else
        Begin grill
   end  end
else
   Begin
     if (tall) then
        Begin chop, fry
        end
     else
        Begin
          if (juicy) then
             Begin boil
             end
```

```
  else
     Begin roast
     end
 end
end
```

This form has been shown by research to be very effective in reducing programming problems (number of errors, maintainability or ability to change the program readily, etc.). However, it is not without flaws. For one thing, the program has doubled in length, making it longer and perhaps harder to memorize quickly. Notice that when you get to an else, you have to look around to find out what conditions led you to the else. Was it because it was not leafy or not green, etc.? Some researchers investigating this problem have proposed the following:

```
if (green) then
   (block of code 1)
else
   (block of code 2)
```

Should become

```
if (green) then
   (block of code 1)
if (not green) then
   (block of code 2)
```

An analogous problem occurs when you encounter an END statement. What set of conditions led you to the END? Various languages deal with all of the above problems (in Pascal, you may put a comment after the end to indicate how you got there). However, the teacher should recognize the problems associated with the conditional and some of the possible solutions. It will certainly provide more perspective when dealing with the topic.

Another solution that could be considered (almost all you have to do is to create the compiler to handle the situation) is the following:

```
if hard peel
   if green roast
   not green grill
   end green
not hard
   if tall chop, fry
   not tall
      if juicy boil
      not juicy roast
      end juicy
   end tall
end hard
```

The teacher can discuss problems with this form as well. What happens if instead of "peel" in the above program you require ten statements and so on? The point is that there are many formats for the conditionals in many programming languages and that no one explicit form has been recommended above others at this time.

Again, we emphasize that the idea of structured programming is violated by the GOTO. What is desirable is that when you come to a block of code, you enter at the top and leave at the bottom and this benefit is lost through the poor use of a GOTO. Restricted entry and restricted exit to a block of code facilitates the combining of statements that carry out a particular objective, and facilitates comprehension by limiting program complexity.

Earlier in the notes, we commented on the nesting of "if" statements and that a nesting to a depth of more than two is to be avoided where possible since it makes following the program logic more difficult. However, suppose that there is no way around this dilemma and we end up with the following program code:

```
if leafy
   call fry
else
   if hard
      if green
         if round
            call bake
         else
            call chop
      else
         call broil
   else
      if juicy
         call slice
      else
         call throw away
```

The problem here is how did you get to "call chop"? It turns out that the food was not leafy, was hard and green, and was not round. Another way of expressing the above conditions is to use compound logical (or Boolean) expressions:

```
if leafy call fry
if hard and green and round call bake
if hard and green call chop
if hard call boil
if juicy call slice
call throw away.
```

Notice that the above example assumes that after one "IF" statement executes, then the program finishes executing. This is not usually the way the "if" statement is implemented. A full resolution follows:

```
if leafy call fry
if not leafy and hard and green and round call bake
if not leafy and hard and green and not round call chop
if not leafy and hard and not green call boil
if not leafy and not hard and juicy call slice
if not leafy and not hard and not juicy call throw away
```

Notice that this does not seem more appealing. Hence, when the teacher comments that nesting of "if" statements to a depth of more than two is unattractive, then students should be aware of what some of the possible alternatives are.

The teacher can conduct many of these logic exercises in pseudo-code since it is quicker and easier than preparing a program and executing the code.

Note that another conditional statement is the case construction. Its advantage is that a multi-way branch is accomplished in a single, well-organized statement which has an explicit end statement.

The computed GOTO carries out the case but it is not organized and has no explicit end. Only a statement within a case is under the case control. However, with a computed GOTO, if it is not used in a controlled, organized fashion, it is possible that any statement in the program might be executed.

APPENDIX D

SUGGESTIONS FOR TESTS OR FOR ASSIGNMENTS

Further to suggestions made for evaluation on page 6, the teacher may consider the following:

1. Give students a program and ask them to modify it, either to accomplish a new task or to get it working.

   Example

   Given a program that works on a vector, modify the program so that it increases each value in the array by some amount.

   Often, on tests students write programs that are almost working. These form the basis for a good assignment (or a later test), where other students are asked to get the program working by modifying it (possibly with as few modifications as possible).

   Other questions could be created from the following general points:

   1. What is the value of a variable at a specific point in the program?
   2. What is the sequence of values assumed by a variable?
   3. What is the number of times a particular statement or sub-program is executed?
   4. Give the output for a given input?
   5. What is the function of a variable? What is the function of a statement or an entire program? Describe it briefly.
   6. What is the impact of a change in the program?

2. Find a program in a textbook that is at the approximate level for the students. Let the students study the program for, perhaps, eight minutes, and then ask them to reconstruct the program. The more experienced or better the programmer, the better will be the reconstruction performance.

   What may be happening is that as students gain experience in programming, their ability to recognize meaningful program structures (loops, etc.) is improving, enabling them to recode the syntax of the program into a higher-level interval semantic structure, and thus enabling them to remember the program.

   It is suggested that performance on a reconstruction task is a good measure of program comprehension. A student's ability to rote-memorize something as complex as a computer program is most difficult. However, a knowledgeable programmer can successfully recode program statement groups into even higher-level semantic structures. A student's ability to reconstruct the program indicates that he/she understands the low-level details of each statement, intermediate level groupings of statements, and the overall purpose of the program. This last statement could also serve as the best definition of program comprehension: to be able to recognize the overall purpose of a program and to understand the intermediate-level processes including program organization and an understanding of each program statement.

   Research also suggests that the syntax of each statement is only briefly retained and only the underlying semantic structure is stored in long-term memory. As well, information on each individual statement is not stored by the student one statement at a time, but rather the student carries out a recoding process to a higher-level semantic structure which captures the meaning of a group of program statements.

Hence, simply asking a student to memorize a program and then reconstruct it, is a convenient testing technique. Interesting variations on this theme follow:

a) Ask students to give a written description of the purpose of a program and its organization.
b) Give students parts of the program which they were asked to memorize and then ask them to supply the missing statements.

c) Provide multiple-choice questions on the program to be memorized.

The task of memorizing and then reconstructing a program is not only an aid to help a student comprehend programs in general, but is also a convenient testing tool.

Note: Experienced programmers might write programs that do essentially the same task but that are syntactically different. For example, they may use different variable names consistently, or change the order of the statements where it did not matter. The teacher may decide to permit this as a correct response.

APPENDIX E

SUGGESTIONS FOR USE OF PROGRAM COMMENTS

1. Types of Comments

a) High-level comments are placed at the beginning of the program or group of statements and are functionally descriptive, indicating what a group of statements do.

Example: A program reads an integer N which tells how many data statements follow. The N data statements containing student grades are read into the array G. The class average is found and the number of the student whose grade deviates the least from the average is printed with his/her grade.

b) Low-level comments reiterate the opertion of a particular statement.

Reserve 500 locations for the G array
Loop from 1 to N
Set G1 to the first value of G
Compare DEVGJ to DEV
Write out J and GJ

2. Advantages and Disadvantages

When one reads a program, one would like to understand and to remember what it accomplishes. The mind recodes the detailed program information into higher-level semantic (meaning) structures. The high-level comments assist this process and make the program easier to recall.

The low-level comment for the statement 1 = 1 + 1 is ADD ONE TO THE VARIABLE 1 and is a bad comment. What is needed is a statement that gives meaning to a group of program statements. For example, "Search for the largest value in the table." This can be carried further to make it a better comment by relating the comment to the program that is being solved. In the previous example, the last comment becomes "Find the student with the highest term average."

Comments can have negative effects, including:

a) disruption of the visual scanning of the program;
b) lengthening the program so that pages have to be turned in order to see the entire program.

These effects often cause programmers to remove comments when studying programs or to leave them out until a program is debugged.